On the Implementation of Polygonal
Approximation Algorithms

Ovidiu Grigore
Remco C. Veltkamp

institute of information and computing sciences, utrecht university
technical report UU-CS-2003-005
www.cs.uu.nl
1 Introduction

The polygonal approximation, as a simplification procedure of an input curve representation, consists of the main subject of many studies, but most of them are exclusively focused only on the theoretical matters and the practical aspects about their implementation are usually ignored. From theoretical point of view the main goal is to find new algorithms for curve polygonal approximation, thus, using new ideas, new data structures or new searching techniques, always the final challenge is to improve the performances of the existing methods. To proof the superiority of a new algorithm, theoretical upper bounds of parameters like algorithm (running time) complexity and used memory are established and compared between the algorithms.

These studies, which point out only the theoretical aspects of an algorithm, do not take into account the practical behavior of a method or the effort needed to implement it and therefore, in these cases someone could say that the algorithm is not completely described. Only a theoretical approach about a method does not guaranty its right behavior in practice and more over, the most of them do not explain how to implement the algorithm in practice.

The principal aim of thinking, designing and developing a new algorithm is its practical usage. There are methods with poor theoretical performances that are easy to be implemented and, on the other side, there are algorithms with very good theoretical features for which the great effort of practical implementation makes them almost unusable. Also, there are different algorithms that have the same theoretical performances, but they can be ordered according with either the effort needed to implement them or their practical efficiency. Therefore, besides the theoretical description and evaluation, any algorithm should be analyzed also from the experimental point of view.

This study tries to cover the lack of experimental details among several theoretical polygonal approximation algorithms. Three main directions are followed in this experimental analysis:

- the possibility of an efficient implementation of the algorithms and the effort needed to do it;
- the comparison of the algorithms using experimental measurements;
- the correspondence between the theoretical feature upper bounds and the experimental results.

To fulfill this task, the first step was to design a software package that implements several algorithms used in polygonal approximation of an input curve. For this purpose there are applied optimal methods, like dynamic programming [8], and sub-optimal approaches, like recursive split [6]. Both, bounded-# and bounded-ε, approaches were studied and implemented. All these algorithms were implemented in a generic manner, trying to make the software package as general as possible. Thus, depending on the application, someone can chose between different items that are possible to be set for features like:

- the base data type (integer, double, leda_integer, Gmpz, etc.),
- the kernel representation (Cartesian, Homogeneous),
- the distance used in measuring of the approximation error.

To overcome all these requirements, the concept of traits class was used.
A library of traits classes implementing different kinds of distances between a point and a segment is available. There are implemented common distances like: the Euclidean distance, the Manhattan distance, the $L_\infty$ distance, measured to the line support or to the segment, but also other distances, like the bounding shell distance \cite{3}. Also, new computational strategies useful in improving the running time, like the incremental algorithm used to compute the Euclidean distance \cite{4}, or the algorithm based on the path hull structure \cite{1}, are implemented.

In the same spirit of generalization, trying to cover as many as possible concepts, the approximation error can be assessed either as the maximum value of all the distances computed between the input curve and the output polygonal approximation, or as the sum of all these distances.

Another group of algorithms, optimal polygonal approximation methods for particular distance measures, was implemented. The generic character regarding the base data type and the kernel type is still available, but these methods use graph search techniques \cite{5} to compute the polygonal approximation with the Euclidean distance error assessment only.

After an introduction about the purpose of this study, in the second section are presented some theoretical elements about polygonal approximation methods and the optimal and sub-optimal paradigms are explained. After these, different kinds of distances used in polygonal approximation error assessment are introduced. Also, there are described specific techniques used to improve the performances of the polygonal approximation algorithms for some particular cases, like the incremental technique used for computing the sum of the squared Euclidean distances or the path hull structure used to improve maximum Euclidean distance assessment for the Recursive Split method.

The software package is described in section three. The structure and the concepts used to implement the polygonal approximation methods are discussed. Also a comparison between the generic and the dedicated implementation techniques is briefly reviewed.

In the last section the experimental results are presented. The algorithms are compared from practical measurements point of view. The steps made to improve the polygonal approximation algorithms both from theoretical and practical perspectives are presented. Conclusions that concern different correlations between the experimental results, like comparison between the different algorithms, comparisons between different kind (generic and dedicated) of implementations and comparison between the theoretical bounds and practical results are given.
2 Theoretical Elements

2.1 Definitions

Let \( \{P_1, \ldots, P_N\} \) be a set of successive points, defining the input curve, then, with respect to a given criterion, it can be represented simplified as a set with a reduced number of points \( \{Q_1, \ldots, Q_M\} \), \( M < N \).

\[
\begin{array}{c}
\text{Input curve} \\
\text{Polygonal approximation}
\end{array}
\]

![Fig. 1 Example of polygonal approximation.](image)

To define the differences (roughness) between the input curve and its approximation (the output curve), two parameters can be used: the decrease in the number of points used in curve representation (\( N-M \)) and the error of approximation that measures the changing of the input curve. Depending on the main parameter of interest, there are two possible approaches that can be used for solving a polygonal approximation problem:

- **min-\#**, when given an error threshold that is the upper limit of the distortion accepted in the final representation, the polygonal approximation with the minimum number of points has to be computed (figure 2);

- **min-c**, i.e., given the number of points desired to be used in the final representation, the output curve that approximates as good as possible from a given error representation criterion point of view has to be found (figure 3).
2.2 Approximation Error Assessments

For both categories of algorithms an evaluation of the error appeared in approximation of the initial curve is needed. To estimate it, several distances from a point to a line can be applied, such as: the Euclidean Distance, the Manhattan Distance, the $L_{\infty}$ distance, the Vertical Distance, the Bounding Shell Distance.

Also, the error obtained in approximation of a curve fragment with a line segment can be defined either as the maximum value of the distances between the segment and each point of the curve portion or as the sum of these distances.

One main group of distances consists of particular cases of the general distance $L_p$. For two points $v$
and \( w \) from a \( k \)-dimensional space it is defined as:

\[
L_p(v,w) = \left( \sum_{i=1}^{k} |v_i - w_i|^p \right)^{\frac{1}{p}}
\]

where \( v_i \) and \( w_i \) are the \( i \)-th coordinate of the \( v \) and \( w \), respectively.

### 2.2.1 The Manhattan Distance (\( L_1 \))

For \( p=1 \) the distance obtained is usually called Manhattan distance or city block distance. For a \((x,y)\) plane, it is defined as:

\[
L_1(v,w) = |v_x - w_x| + |v_y - w_y|
\]

This distance is equal to the length of the shortest path between the two points while going only along the axis directions. In figure 4 is presented the measure of the Manhattan distance between a point and a segment.

![Manhattan distance measurement](image)

**Fig. 4** Manhattan distance measurement \((d_x+d_y)\).

### 2.2.2 The Euclidean Distance (\( L_2 \))

Maybe the mostly used distance is the Euclidean one, obtained from the general formula for \( p=2 \):

\[
L_2(v,w) = \sqrt{(v_x - w_x)^2 + (v_y - w_y)^2}
\]

In figure 5 the distance between a point and a line is shown.
2.2.3 The Max Distance ($L_\infty$)

$L_\infty$ is the particular case obtained for $p \to \infty$, when, for a $k$-dimensional space, the formula becomes:

$$L_\infty(v, w) = \max_{i=1..k} \{ |v_i - w_i| \}$$

where $v_i$ and $w_i$ are the $i$-th coordinate of the points $v$ and $w$, respectively.

As is easy to observe, to measure the $L_\infty$ distance is similar to the $L_d$ procedure (fig. 4), but this time only the maximum value of the distances computed along the axis is taken into account.

2.2.4 The Vertical Distance

The vertical distance is equal to the difference in $y$-values between the current point and the intersection point of the vertical line that is passing through it and the line support of the approximation segment. Figure 6.a. shows an example of measuring vertical distance, when the intersection point is inside the segment and in figure 6.b, is presented the case when the intersection point is outside the segment.

---

Fig. 5 Euclidean distance measurement ($d_{\text{eff}}$).

---

Fig. 6 Vertical distance measurement.
Please note that when this distance is used to assess curves that contain vertical parts, these cannot be practically approximated. In that case the vertical parts are returned unchanged.

2.2.5 The Bounding Shell Distance

The *shell* is a bounding volume for the curve segment that has to be approximated. It consists of the two smallest circle arcs passing through the two end points of the approximating segment that includes between them all the points of the curve portion to be approximated. If all the points are on the same side of the approximating segment then the bound on the side without points is the segment itself. The distance associated to a bounding shell is defined as the maximum value of the Euclidean distances measured between the segment and all the points of the two bounding arcs [3].

![Diagram of Bounding Shell](image)

**Fig. 7** Finding the bounding volume and its associated distance.

2.2.6 The Path Hull Based Distance

The *convex hull* of a set of points is the smallest convex set containing it. It can be proved that the furthest point from any segment linking two points of the initial set is always an element of the convex hull.

To build a polygonal approximation using a recursive split method based on the maximum Euclidean distance means to find in each step the furthest point from a certain segment. Therefore, only the points from the convex hull of the current curve need to be considered to find the furthest point.

On applying the recursive split method, the current curve is changing, actually in each step a sub-chain of the initial curve is processed. To deduce efficiently the convex hull of the current sub-chain using the information from the previous step of processing, an efficient data structure, called *path hull*, was developed [1].
A path hull consists of two arrays storing the vertices of the two convex hulls: the left stores and the right stores. Attached to each array is a *history stack*, used to restore the previous hull, when this operation is needed.

This method uses the *path hull* structure [1] to compute the Euclidean distance measured to the line support of the approximation segment (see chapter 1.3.6, page 8). The main idea of this algorithm is to use the convex hull of a curve to find faster the maximum distance between the curve and an approximation segment. There are two possible cases in obtaining the current convex hull: first, to *recover* it from the previous one and the second, to *build* it without using any information from the previous processing. When the *recover* procedure is chosen, the operations used to obtain the current hull, which were stored in the *history stack*, are applied in reversed order to restore the previous hull.

To improve the practical performances, the *build* operation is implemented using the on-line convex hull algorithm [9].

### 2.2.7 The Incremental Technique Based Distance

The assessment performances for some particular distances can be improved in certain situations applying specific computational algorithms. The incremental technique described by Perez and Vidal in [4] is such an algorithm designed to improve the computation of the sum of the squared vertical or Euclidean distances. Let \( \{ p_0, p_1, \ldots, p_n \} \) be a set of consecutive points representing the input curve. The incremental method can be applied to approximate subsets of consecutive points \( \{ p_b, p_{b+1}, \ldots, p_j \} \) by the segment \( p_b p_{j+1} \) linking their ending points. The constraint is that the subsets must have the same starting point \( p_b \) and they are extended successively, point by point, in only one direction \( (p_b, p_{b+1}, \ldots) \).

For the sum of the squared vertical distances the error of approximating the curve portion \( \{ p_b, p_{b+1}, \ldots, p_j \} \) with the segment \( p_b p_{j+1} \) is computed as:

\[
svd(i, j) = \sum_{i=1}^{j} ((a_{i,j} + b_{i,j} x_i) - y_i)^2
\]

where:
- \( a_{i,j} = y \cdot b_{i,j} \) is the y-value of the line \( p_b p_{j+1} \) at the y-axis.
- \( b_{i,j} = \frac{y_j - y_i}{x_j - x_i} \) is the direction coefficient of the line \( p_b p_{j+1} \)

If the function \( svd(i, j) \) needs to be computed for the same set plus a new point, as it was stated above, than the new approximation segment will share one endpoint with the last one, but its other endpoint will be different. Thus, the direction of the new segment is different from the previous one and therefore, in a classic manner of computation, the value of \( svd(i, j) \) can be assessed only by evaluating again all the distances from the points to the new segment. Fortunately, this expensive procedure can be avoided using the incremental technique that is based on the following formula obtained by expanding (1):

\[
svd(i, j) = a_{i,j} (j-i-1) + 2a_{i,j} b_{i,j} \sum_{k=i}^{j} x_k - 2a_{i,j} \sum_{k=i}^{j} y_k + b_{i,j}^2 \sum_{k=i}^{j} x_k^2 + \sum_{k=i}^{j} y_k^2 - 2b_{i,j} \sum_{k=i}^{j} x_k y_k
\]

If five variables holding the sum of the \( x \)'s, \( y \)'s, squares of \( x \)'s, squares of \( y \)'s and the cross products \( xy \) are incrementally maintained, than computing the new total error \( svd(i, j+1) \) involves only updating these variables with the contribution of the \( j+1 \)th point, which takes constant time. Also, in each step the parameters \( a_{i,j} \) and \( b_{i,j} \) defining the new segment must be computed.
To compute the sum of the Euclidean distances $sed(i,j)$ in the same conditions, the incremental technique can be applied in a similar way according to the following formula describing its dependence on the sum of the vertical distances:

$$sed(i,j) = \frac{1}{b_{i,j} + 1} vsd(i,j)$$

### 2.2.8 Distance Measured to the Segment or to the Line Support

Whenever the procedure of measuring the distance between a point and a segment implies the projection of the point on the segment, there are two possible cases: either the projection point is inside the segment or it is outside. Therefore, two kinds of distances can be assessed:

a) the distance measured to the line support of the segment: in this case the distance is computed always between the point and its projection, even if it is outside of the segment, on its line support (fig. 8);

b) the distance measured to the segment: this time two cases can happen: first, when the projection of the point is inside the segment and the distance is computed similarly with case (a); in the second approach the projection is on the line support, outside the segment, so it is chosen the closest point from the segment, actually the closest end point of the segment, to the current point (fig. 8).

![Diagram](https://example.com/diagram.png)

*Fig. 8* Distance measured to the line support or to the segment. The difference between the two approaches is presented for the Euclidean distance.
2.3 Polygonal Approximation Algorithms

2.3.1 Optimal and Sub-Optimal Solutions

Two parameters characterize the polygonal approximation of an input curve: its number of vertices \( M \) and the error \( \epsilon \) that approximates the input curve. Setting an upper bound for one of these parameters, the value of the other one depends on the algorithm that is applied, but the smaller the value the better the algorithm. So, from this point of view, the polygonal approximation methods can be split in two main groups:

- the bounded-\( \epsilon \) algorithms, when the upper bound of the approximation error is given
- the bounded-\# algorithms, when the upper bound for the number of vertices of the output poly-line is demanded.

According to the accuracy of their solution, we can roughly divide the algorithms used to approximate polygonal curves into two main categories: the optimal and the sub-optimal methods, respectively.

In the former case, optimization techniques, like dynamic programming, are applied to find the best solution of the approximation problem. Therefore, in this case, the bounded-\# problem is equivalent with a min-\( \epsilon \) approach (section 1.2, page 4) and the bounded-\# with the min-\# (section 1.2, page 5).

Generally, these algorithms have relative slow running time, even though there existed attempts to implement faster optimal algorithms for some specific error measurements, such as the incremental technique made by Perez and Vidal [4] for the Euclidean and vertical distances measured to the line support of the segments.

The latter group of algorithms tried to be a compromise between the loss of precision of the solution and the improvement of their running time. Usually, they are based on heuristic techniques, such as the recursive split method, that offers a faster running time, but the solution is not so good as for the optimal one.

<table>
<thead>
<tr>
<th>Algorithm and implementation specifications</th>
<th>Time complexity of computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursive Split, [6]</td>
<td>( O(N^2) )</td>
</tr>
<tr>
<td>General implementation</td>
<td></td>
</tr>
<tr>
<td>Recursive Split,</td>
<td>( O(N\log N) )</td>
</tr>
<tr>
<td>Maximum Squared Euclidean Distances, Path Hull implementation [1]</td>
<td></td>
</tr>
<tr>
<td>Dynamic Programming, [8]</td>
<td>( O(MN^2) )</td>
</tr>
<tr>
<td>General implementation</td>
<td></td>
</tr>
<tr>
<td>Dynamic Programming, new implementation</td>
<td>( O(N^2) )</td>
</tr>
<tr>
<td>General implementation</td>
<td></td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td>( O(MN^2) )</td>
</tr>
<tr>
<td>Sum of Euclidean Distances, Incremental technique [4]</td>
<td></td>
</tr>
<tr>
<td>Graph Search Method, [5]</td>
<td>( O(N^3) )</td>
</tr>
<tr>
<td>Maximum of Euclidean Distances, measured to line support</td>
<td></td>
</tr>
<tr>
<td>Graph Search Method,</td>
<td>( O(N^3) )</td>
</tr>
<tr>
<td>Sum of Euclidean Distances, measured to line support</td>
<td></td>
</tr>
</tbody>
</table>
To understand the difference in running time of the several kinds of algorithms, in table 1 (page 12) are presented the time complexity limits for some of them. \(N\) denotes the number of points of the input curve and \(M\) represents the number of points of the output approximation. The usual implementation cases, which don't use special structures or techniques, are named in table by "general implementation".

### 2.3.2 Dynamic Programming Method

Dynamic programming is a technique used in solving optimization problems, this time applied to finding the optimal polygonal approximation for a given input curve and the desired parameters. Generally speaking, its main idea is to build the final optimum by finding optimum solution at the partial problems. It finds first the optimum solution of the simplest possible level of the problem. Each time the next level is more complex than the previous one, but computing the optimum solution of the current level is based on the optimum solutions found in all bottom levels. In the final level, when the partial problem overlapped the initial problem, the global optimal solution is found.

In the case of polygonal approximation problems, the dynamic programming technique starts with computing all the approximations of all possible parts of the input curve using only one segment:

\[
\{ \Delta(l, n) \}_{l=1..N}
\]

After this, step by step, the approximations of the curve using an increasing number of segments (2, 3... \(M\)) are computed using an iterative process based of the following formula:

\[
D(n,m) = \begin{cases} 
\Delta(l,n) & n \geq m = 2 \\
\min_{i \in \{m+1, \ldots \}} \{D(i,m-1)+\Delta(l,n)\} & n \geq m > 2
\end{cases}
\]

### 2.3.3 Recursive Split Method

This algorithm is a heuristic approach, finding a sub-optimal, but often good, solution of the approximation problem, the main aim being the improvement of the other features of the optimal algorithms, like running time and the memory usage.

Actually, recursive split method is based on a "divide and conquer" technique. It starts with the initial segment that links the two end points of the input curve, for which the furthest point is found. If the distance from the furthest point to the segment is greater than a given error threshold, then the problem is split into two similar sub-problems: approximating the two curve portions, one from the start point to the furthest point, the second from the furthest point to the end point. So, the algorithm is repetitively applied until all the segments obtained approximate their curve portion with an error less than the given threshold (figure 9).
2.3.4 Graph Search Method

In this case an optimal solution of a polygonal approximation problem is obtained applying searching techniques to find the shortest path in the distances-graph associated to the input curve.

The distances-graph $G(C)$ associated to an input curve $C$ given as an ordered set of points $\{p_0, p_1, ..., p_N\}$, is that graph having a vertex $v_i$ corresponding to each point $p_i$ and an arc $(v_i, v_j)$ for each pair of points $(p_i, p_j)$ of the input curve. For a given bound error $\varepsilon$, the graph $G(C, \varepsilon)$ is defined as the subgraph of $G(C)$ containing only those arcs with the weight less than or equal to $\varepsilon$. Any path in $G(C, \varepsilon)$ that starts with $p_0$ and ends with $p_N$ corresponds to a polygonal approximation of the input curve with the maximum error equal to $\varepsilon$.

In the bounded-\(\varepsilon\) (min-\#) approach, an approximation of the input curve having the minimum number of vertices corresponds to the shortest path from $v_0$ to $v_N$ in $G(C, \varepsilon)$, where the length of the path is computed as the number of its arcs. The shortest path can be found using a modified breadth-first search technique, which takes time linear in number of arcs from $G(C, \varepsilon)$. The breadth-first search technique is changed to process the points only in the forward direction, from the first point $p_0$ to the end point $p_N$, otherwise it will be possible to obtain false loops in the resulting approximation polygon (figure 10).

![Image of a polygonal approximation](image-url)

**Fig. 10** Appearance of a false loop in the polygonal approximation of a curve when the classic breadth first search technique was used.
In this approach the solution of the bounded-\( \# \) (min-\( \varepsilon \)) problem is to find the minimum \( \varepsilon^* \) such that its associate graph \( G(C,\varepsilon) \) contains a path from \( v_1 \) to \( v_N \) with the length equal to the desired number of points for the output polygonal approximation \( M \). For doing this, first, the set containing all the possible approximation errors of the input curve is determined. The second step is to find the minimum value \( \varepsilon^* \). Before this, it is interesting to observe that:

- for any possible value \( \varepsilon_1 < \varepsilon^* \) the minimum number of points \( M_1 \) that approximate the curve with this upper bound is greater than or equal to \( M \):
  \[ M_1 \geq M \]

- for any possible value \( \varepsilon_2 > \varepsilon^* \) the minimum number of points \( M_2 \) that approximate the curve with this upper bound is less than or equal to \( M \):
  \[ M_2 \leq M \]

Thus, using binary search among the set of all possible errors, the minimum value \( \varepsilon^* \) is found in \( N \log N \) steps. In each step of the binary search, the minimum number of points necessary to approximate the curve is computed applying the min-\( \# \) algorithm for the current error value.
3 Software Description

The main idea used in designing the software package is to obtain a code as general as possible, that will give it a flexibility that makes it adaptable to as many as possible user requirements. The software developed for this purpose tries to meet all these requirements, but also to combine all the theoretical concepts described before (sections 2.1-2.3).

The structure of the software package is described in figure 11. Thus, the first step is to choose the Base data Type. This data type is identical with the number representation and it will be used directly in computation of the polygonal approximation if in the second stage the Cartesian Kernel is chosen, or it is the base type, the type of the numerator and of the denominator, of the rational number representation when the Homogeneous Kernel will be used. After these, two main categories of functions can be applied to obtain the polygonal approximation. The first one is a more flexible designed software, in which, based on the traits class concept, several distances used in error assessment might be passed to the generic implemented polygonal approximation functions. The second group consists of more dedicated implementations, meaning polygonal approximation functions based on algorithms designed only for a certain kind of distance (Euclidean distance measured to the line support).

**Fig. 11** The structure of the software package.
The polygonal approximation software package is structured as it follows:

- the file "polyap.flt" that contains 10 functions:
  - 6 functions from the generic implementation category used to approximate polygonal curves, covering both the bounded-\# and the bounded-\(\epsilon\) approaches and implemented in optimal (dynamic programming) and sub-optimal (recursive split) manner. Being implemented with templates and traits classes, these functions can be used with several kinds of distances.
  - 4 dedicated functions implemented algorithms based on graph search techniques, solving the min-\# and the min-\(\epsilon\) problems, using as error assessment the sum of or the maximum of the Euclidean distances measured to the line support. Implemented with templates, all the functions can be used with several kinds of number types and several kinds of input and output storing data structures.

- the file "polyap_traits.ll" that contains distance traits classes. Also these functions can be used with different kinds of kernels and with different kind of number types.

3.1 Functions Description

To be as general as possible, the functions deal with templates to define the input and the output data storing structures and also a traits class to specify the distance type used in error evaluation. Thus, the general requirements for the polygonal approximation functions are:

**InputIterator**, which is actually a forward iterator corresponding to the data structure storing the input curve;

**OutputIterator**, similar to the former one, but associated to the result storing data structure;

**DistTraits**, the traits class that defines the distance method used in polygonal approximation assessment. It has also to provide several other items, as follows:

```cpp
typename DistTraits::DataT, a number type;
```

3.1.1 Dynamic Programming Functions Specific Requirements

```cpp
DataT operator()( InputIterator begin, InputIterator end )
```

an operator that returns the error obtained when the curve portion between the begin point and end point is approximated by a line segment

```cpp
DataT cumulative( DataT curveError, DataT deltaError )
```

function that computes the error of a curve segment when two partial errors, curveError and deltaError, of its disjoint parts are known. Usually, this can be done either by taking the maximum value of them or by summing them.
3.1.2 Recursive Split Functions Specific Requirements

```cpp
DataT operator() ( InputIterator begin,
    InputIterator end,
    InputIterator* p_MAX)
```

an operator that, like the previous one, returns the error obtained when the curve portion between the `begin` point and `end` point is approximated by a line segment. The difference is that this time it has to return, through the parameter `p_MAX`, the pointer to the furthest point of the curve from the segment line approximation.

3.2 Distance Traits Classes Description

As mentioned above, to apply the polygonal approximation functions from the generic implementation category, there must be specified a distance traits class. Someone can implement his own traits class or can use those from file `polyap_traits.h`, but the distance traits class has to agree with (obey to) all the requirements stated before in chapter 1.5.1 (see page 11).

All the distance traits classes that are already implemented and are accessible from the file `polyap_traits.h`, use a template class to specify the kernel type, *Cartesian* or *Homogeneous*. Also, through the kernel we induce a base number type.
4 Experimental Results and Conclusions

4.1 The Dynamic Programming Implementation

4.1.1 Original Algorithm

Dynamic programming is an optimization technique based on a progressive computation procedure that finds the best solution of a given problem by computing the optimum of its partial problems. It starts by solving the simplest possible sub-problem of the initial one and evolves step by step, each time determining the more and more complex sub-problems optimum, until it finds the final solution, which actually corresponds to the given problem.

In the case of polygonal approximation problems, the dynamic programming technique starts with computing all the approximation errors of all possible parts of the input curve using only one segment:

$$\{\Delta(i, n)\}_{n=1,N}$$

After this, step by step, the approximations of the curve using an increasing number of segments (2, 3...M) are computed using an iterative process based on the following formula:

$$D(n,m) = \begin{cases} 
\Delta(1,n) & n \geq m = 2 \\
\min_{i=m}^{m} \{D(i,m-1) + \Delta(i,n)\} & n \geq m > 2
\end{cases}$$

The initial algorithm, given in [8], that uses the function $\Delta(i, j)$, computes the error of approximating all points between two points $i$ and $j$, each time needed in partial optimum evaluation, even if this is a redundant operation. The algorithms solving the bounded-$\varepsilon$ (min-#) and the bounded-$\#$ (min-$\varepsilon$) polygonal approximation paradigms are similar, therefore in the following only the algorithm for the bounded-$\varepsilon$ (min-#) approach is presented and analyzed, but the procedures and the conclusions can be easily extended also to the bounded-$\#$ (min-$\varepsilon$) case. The input curve is assumed to have $N$ points and the desired upper bound of the error approximation is $\epsilon_{si}$ and the resulting number of points of the output polygonal approximation is $M_{Out}$.

Dynamic Programming bounded-$\varepsilon$ (min-#) Algorithm (DynProg-E-OLD)

```plaintext
// Compute the optimum polygonal approximation
for n:=2 to N do
    ApproxError$_{n,2} = \Delta(1, n)$
for m:=2 to N do
    for n:=m to N do
        ApproxError$_{n,m} := \min_{i=m}^{m} \{\text{ApproxError}_{i,m-1} + \Delta(i,n)\}
```
for i := n-2 to m-1 do
    dist := ApproxError_{i+1} + Δ(i, n)
    if dist < ApproxError_{m} then
        ApproxError_{m} := dist
        SplitPoints_{m} := i
    if ApproxError_{m} ≤ epsi then
        N_out := m
        break

// Save the polygonal approximation, put the points in the correct order
container.add(Point_i)
for m := N_out to 3 do
    k := splitPoints_{m}
    container.add(Point_i)
container.add(Point_i)

The order of the time complexity of this algorithm is \( O(MNP) \) and the used memory is \( 2NP \), equivalent with \( O(NP) \).

The last part of the algorithm is used to save and to order the points of the output polygonal approximation. To put the points in the correct order, an \( N \times M \) matrix, \( \text{SplitPoints} \), containing the splitting points is computed. This part of the algorithm remained unchanged in all the further versions of the algorithm.

In the first part of the algorithm, which computes the optimum approximation, an \( N \times N \) matrix, \( \text{ApproxError} \), is used to store all the approximation errors obtained when parts of the input curve are approximated with \( m \leq N_{\text{out}} \) points.

4.1.2 Improved Algorithm

The first idea for improving this algorithm was to eliminate the redundant computation of the distances \( Δ(i, j) \) in each step of the optimum searching part of the method. For developing this, the initial loop used to compute the optimum solution is split in two parts:

1. the computation of all the distances between pairs of points from the initial curve and storing them in an extra data structure, which is equivalent to construct a complete graph associated to the input curve, having the points as vertices and the distances between them as label of the its arcs;

2. the search of the optimum approximation using the distances computed in the first step, which is equivalent to find the best path in the distances graph according to a given criterion.

In such circumstances the new algorithm is:
Dynamic Programming bounded-ε (min-#) Algorithm (DynProg-E-NEW)

// Distances graph construction
for m:=2 to N do
    for n:=m+1 to N do
        Dist_Graph_{m,n} := Δ(n,m)

// Searching the optimum polygonal approximation
for n:=2 to N do
    ApproxError_{n,2} = Dist_Graph_{n,N}

for m:=2 to N do
    for n:=m+1 to N do
        ApproxError_{n,m} := ApproxError_{n,m-1} + Dist_Graph_{n,n}
        for i:=n-2 to m-1 do
            dist := ApproxError_{i,m-1} + Dist_Graph_{i,n}
            if dist < ApproxError_{i,m} then
                ApproxError_{i,m} := dist
                SplitPoint_{i,m} := i
        if ApproxError_{n,m} ≤ ε then
            M_out := m
            break

// Save the polygonal approximation; put the points in the correct order
container.add(Point_{n})

for m:=M_out to 3 do
    k := SplitPoint_{i,m}
    container.add(Point_{k})
    container.add(Point_{i})

In this new approach, in the worst case, the used memory increases from 2Δ^2 to 3Δ^2, so it remains of order O(N^2). The advantage of the new implementation is the decrease in time complexity, from O(MN^3) to O(N^3).

4.1.3 Theoretical vs. Practical Time Complexity

Experimental running time measurements made using the old and the new implementation are presented in figure 12. Data and experimental conditions are given in annex, table 1, page 56. The difference in running time between the two implementations is obvious, but moreover, the practical results obtained for the new implementation confirm the theoretical expectation, the running time in this case is constant in M.
In the following, a detailed analysis regarding the running time of the Dynamic Programming new implementation, both from theoretical and experimental point of view, is presented.

The theoretical running time complexity bound is obtained by finding the values for the two parts that compose the new algorithm:

1. the distances graph construction:
   \[
   t_{\text{dist}} = \sum_{m=2}^{N-1} \sum_{n=m+1}^{N} \Delta(n,m) = \sum_{m=2}^{N-1} \sum_{n=m+1}^{N} \sum_{k=m}^{n} c_t = O(N^3)
   \]

2. the search of the optimum solution:
   \[
   t_{\text{search}} = \sum_{m=2}^{M} \sum_{n=m+1}^{N} \sum_{k=m}^{n-2} c_t = O(MN^2)
   \]

3. ordering the points of the output polygonal approximation:
   \[
   t_{\text{order}} = \sum_{m=1}^{M} c_t = O(M)
   \]

The total time needed to process an input curve is:

\[
 t_{\text{DynProg-NEW}} = t_{\text{dist}} + t_{\text{search}} + t_{\text{order}} = O(N^3)
\]
Important to remark is that the final bound of the time complexity is imposed by the first part of the algorithm in which the distances graph is computed. Therefore, as it will be further shown, any improvement made in the searching part of the algorithm won't change the theoretical bound of the time complexity as long as the distances graph computation method remains the same.

To point out the correspondence between the theoretical problem and the practical implementation, several experiments were made to estimate practical values of the running time and to compare them with the theoretical bounds computed above.

In figure 13 the influence of the output number of points $M$ in running time complexity of the new implementation of the Dynamic Programming algorithm is presented.

The *points* are the practical assessments:

- by *squares* are represented the running time measurements for distances graph construction in the first step of the algorithm
- by *circles* the experimental values obtained in optimum searching step.

The theoretical traces are overlapped:

- the *solid* one represents the theoretical bound for the distances graph computation step, which is constant in $M$
- the *dotted line* corresponds to the optimum searching phase, which is linear in $M$, $O(M)$.

Data and experimental conditions are given in annex, table 2, page 56.

---

![Graph](image)

**Fig.13** The influence of the output number of points $M$ in running time of new implementation of the Dynamic Programming method, bounded-$c$ approach: comparison between theoretical bounds and experimental results.
The second dependence, relating the running time complexity to the input number of points $N$, is shown in figure 14. The experimental results are represented by squares for the distances graph computation phase and by circles for the optimum searching part of the algorithm. Corresponding to these two kinds of practical assessments, the theoretical bounds are: the continuous curve for the distances computation having a $O(N^2)$ complexity and the dotted curve for the searching step of $N^2$ type.

Data and experimental conditions are given in annex, table 3, page 57.

![Graph showing time vs. input number of points $N$.](image)

**Fig.14** The influence of the input number of points $N$ in running time of new implementation of the Dynamic Programming method, bounded-$c$ approach: comparison between theoretical bounds and experimental results.

The general conclusion of this part of experimental and theoretical study is the very good matching between the theoretical expectation and the practical results.

### 4.1.4 Improving the New Implementation of the Dynamic Programming Algorithm

Two modifications of the new Dynamic Programming algorithm are discussed in this section, the first is to improve the memory usage and the second for decreasing the running time. Even if both of them won’t change the theoretical bound of the two parameters, some improvement will be obtained in the practical behavior of the algorithm.

The matrix $\text{ApproxError}$ is used to store the errors of approximation of portions of the initial curve using increasing number of points. The necessity of storing all these values is due to the Dynamic Programming progressive manner of computation that starts with solving simple sub-problems and evolving in each step to a more complex one, in each stage the optimum being computed based on the
previous results. But in a more attentively analysis it will be discovered that 
the computation is 
based only on a first order relationship among the consecutive phases of it, which means that to compute the current optimum value, let's say in step \( m \), only the values computed in the former \((m-1)\) step are needed. So, from the beginning, the initial matrix \( \text{ApproxError} \) can be replaced with two \( N \)-
dimensional vectors, one for storing the optimum values obtained in the last step and the other to keep the values computed in the current step. Moreover, to compute the \( \mathcal{A} \)th value from the current 
step only the component \([1,..., i-1]\) from the former step are needed. Therefore, by changing the order of 
component computation, processing them from the end of the vector to the beginning of it, as it is 
explained in figure 15, the used memory in this stage decreases to one \( N \)-dimensional vector. The 
global used memory is \((2N^2+N)\), instead of \(3N^2\) in the old implementation, but the theoretical bound 
will remain the same \( O(N^2) \).

![Diagram of Processing Order]

Fig.15 The structure of the vector used in the Dynamic Programming new implementation.

The second improvement concerns the running time and derived from adjusting the initial processed 
data structure. So, initially the entire distances graph \( Dist\_Graph(O) \) is taken into account to find the 
optimum polygonal approximation. But in this data structure are stored also paths that corresponds 
to approximations with error greater than the allowed upper bound limit and therefore in those 
cases that include one of these paths the computational process is useless. The solution is to process the 
reduced graph \( Dist\_Graph(O,e) \) containing only the arcs corresponding to a polygonal approximation 
error greater than or equal to the upper bound \( e \).

So, the new and improved version of the Dynamic Programming algorithm is:

Dynamic Programming bounded-\( e \) (min-\#) Algorithm (DynProg-E-NEW_IMP)

```
// Distances graph construction
for m:=2 to N do
    for n:=m+1 to N do
        Dist\_Graph\_m:= A(n,m)

// Searching the optimum polygonal approximation
for n:=2 to N do
    Vect\_Approx\_Error\_n = Dist\_Graph\_n
```

24
for $m := 2$ to $N$ do
  for $n := N$ to $m+1$ do
    VectApproxError$_n := $VectApproxError$_{n-1}$
    for $i := n-2$ to $m-1$ do
      if $\text{Dist}_n \leq \epsilon$ then
        dist := VectApproxError$_{i}$ + Dist$_n$
        if $\text{dist} < \text{VectApproxError}_n$ then
          ApproxErr$_n := \text{dist}$
          SplitPoints$_{n,m} := i$
      if VectApproxError$_n \leq \epsilon$ then
        break
        break
  
// Save the polygonal approximation; put the points in the correct order
container.add(Point$_n$)
$k := N$
for $m := N\text{out}$ to 3 do
  $k := \text{splitPoints}_{n,m}$
  container.add(Point$_k$)
Container.add(Point$_n$)

The average decrease of the running time, computed over the entire set of experimental data presented in table 4 and table 5 in annex (pages 57, 58), is 10% and it depends on the complexity of the reduced graph that is established by the value of the approximation error upper bound $\epsilon$. This experimental result is expected, because the algorithm modification works only over the optimum searching phase and the main part of the time is consumed in the distances graph construction step, which makes the theoretical running time complexity to remain at the same value of $O(N^2)$.

The comparison of the experimental results obtained running the new and the new improved implementations of the Dynamic Programming algorithm, for the particular case of an input error upper bound $\epsilon=1$, is presented in figure 16. The average decrease of the running time, computed only for experimental data measured in this case ($\epsilon=1$), is 20%. A more complete report regarding these experimental data is given in table 4 and table 5 in annex, (pages 57, 58).

Better results are obtained when special distance computation methods are applied to decrease the running time needed to construct the graph in the first stage. This is the situation when the incremental technique proposed by Perez and Vidal in [4] is used to construct the distances graph. In this case the running time complexity of the first stage becomes comparable with de optimum searching step, $O(MN^2)$, and, therefore, the decrease obtained by the new improved implementation in the running time of the searching process is reflected as a better improvement of the entire algorithm.

The entire set of experimental results presented in annex, 4 and table 5 (pages 57, 58), shows that in case of using incremental technique for distance computation, the ratio of the running times measured for the new implementation and the new-improved version ranges between [1.5, 4.9], depending on the reduced-graph simplification degree.
Figure 17 presents a comparison between the running time obtained for the new and the new-improved implementations of the Dynamic Programming algorithm, when the incremental technique is used in the distances graph assessment and for the particular case of an input error upper bound \( \varepsilon = 1 \). For the data measured only in this particular case \( \varepsilon = 1 \), the average ratio representing the decrease of the running time is 4.8. The entire set of the data measured in this experiment is given in table 4 and table 5 in annex (pages 57, 58).
4.1.5 **Generic Implementation vs. Dedicated Implementation**

The software implementing the Dynamic Programming algorithms is designed in a generic manner, being able to deal with different kinds of distance measures, different kinds of data storing structures, different kinds of number types and kernels. The advantages of such an implementation are its generalization and flexibility, but these are paid with the increase of the running time.

To estimate this loss of performance, a dedicated version of software was implemented in such a manner to optimize the new-improved Dynamic Programming algorithm designed particularly for the sum of the Euclidean distances error assessment.

The experimental running time results measured for the generic and dedicated implementations, which are given in Table 6 (annex, page 59), show that the loss in running time because of the generic implementation is ranging between 20% and 67%, depending on the values of the input and the output parameters.

A graphic representation of the difference between the running time of the generic and the dedicated implementations of the Dynamic Programming algorithm is presented in Figure 18.

![Graph showing running time comparison](image)

**Fig. 18** Running time experimental results, comparison between the generic and the dedicated implementations of the Dynamic Programming method: bounded-(min+) approach ($M=78$), using the incremental technique for computing the sum of the Euclidean distances.
4.2 Implementations of the Recursive Split Method

The Recursive Split method is a heuristic technique used to find a sub-optimal, but often good, solution of a polygonal approximation problem. The basic idea of this algorithm consists of repetitively applying of the following steps:

- Approximate the current curve portion with the segment linking its end points. The current curve at the beginning of the algorithm is the entire input curve.
- Using a given criterion, find the splitting point of the current curve and compute the approximation error between the current curve and its approximation segment.
- If the approximation error is greater than a given threshold then repeat the above-explained steps for the two sub-curves resulted by dividing the current curve portion in the splitting point.

Due to the scheme of the Recursive Split method, the simplest way to implement it in a bounded-$\varepsilon$ approach is to use a recursive technique, as it is explained below. The input function parameters are:

- the first point of the curve, $p_1$
- the last point of the curve, $p_n$
- the approximation error upper bound, $\varepsilon_{SI}$
- the list storing the points of the polygonal approximation, poly_list

**Recursive Split Algorithm, bounded-$\varepsilon$ approach, recursive implementation (RecSplit_be_rec)**

```plaintext
RecursiveSplit_be_rec(p_1, p_n, \varepsilon_{SI}, &poly_list)

// compute the polygonal approximation
// of the curve $p_1p_n$ with an error less
// than or equal to the threshold $\varepsilon_{SI}$
{
  d := error(p_1, p_n, &p_split) // the function error returns the error of
  // approximating the curve $p_1p_n$ with the segment $p_1p_n$
  // and also returns the splitting point $p_{\text{split}}$
  if d > \varepsilon_{SI} then
    add_list(p_split, &poly_list)
    RecursiveSplit_be_rec(p_1, p_split, \varepsilon_{SI})
    RecursiveSplit_be_rec(p_{\text{split}}, p_n, \varepsilon_{SI})
}
```

The alternative to this manner of implementation is the iterative technique that does not use the function self-calling. Designing the iterative implementation of the Recursive Split method is a more complex task that requires a supplementary list data structure for storing the segments of the current polygonal approximation of the input curve. In each step of the algorithm the current polygonal approximation is changed and therefore the list structure is dynamically implemented in such a manner to be able to cumulate all the modifications of the approximation polygon with the minimum computational effort.
The basic element of the list, which correspond to a segment of the approximation polygon, stores the following information:

- **begin** corresponds to the starting point of the segment
- **end** represents the ending point of the segment
- **split_pt** is the splitting point of the current curve portion

Also, there exists the **next** field used to link the elements of the list.

With these notions already introduced, the iterative implementation of the Recursive Split method in bounded-\(\varepsilon\) approach is as follows:

**Recursive Split Algorithm, bounded-\(\varepsilon\) approach, iterative implementation (RecSplit_be_it)**

```
RecursiveSplit_be_it(p_i, p_n, \varepsilon) // compute the polygonal approximation
// of the curve p_i\cdots p_n with an error less
// than or equal to the threshold \varepsilon
{
    // initiate the list with the first approximation segment
    // that links the two end points of the input curve
    list_elem := new_element() // generate a new element of the list
    list_elem.next := NULL
    list_elem.begin := p_i
    list_elem.end := p_n
    d := error(p_i, p_n, &p_split) // the function error returns the error of
    // approximating the curve p_i\cdots p_n
    // and also return the splitting point p_split
    list_elem.split_pt := p_split
    current_elem := list_elem // store the current segment
    while d > \varepsilon do
        elem_new := new_element() // generate a new element of the list
        elem_new.next := current_elem.next
        current_elem.next := elem_new
        new_elem.begin := current_elem.split_pt
        current_elem.end := current_elem.split_pt
        d := error(current_elem.begin, current_elem.end, &p_split)
        while current_elem.next && d <= \varepsilon do
            current_elem := current_elem.next
            d := error(current_elem.begin, current_elem.end, &p_split)
        current_elem.split_pt := p_split
    }
```

The confidence in obtaining a better implementation was the reason of studying and testing the iterative version of the Recursive Split algorithm. Actually, a performance improvement in running time was expected after eliminating the function self-calls.
On the contrary, the experimental results, which are presented in Table 2, show that the two implementations, recursive and iterative, are similar from the running time point of view. The explanation of this experimental result is the time needed to process the list data structure from the iterative implementation that is equivalent with the time spent in function self-callings from the recursive version.

The second reason of studying the iterative technique described above was to prepare the implementation of the Recursive Split method in bounded-$\#$ approach. In the bounded-$\epsilon$ all the curve portions are repetitively split until their approximation error becomes less than or equal to the input threshold. Therefore, in this approach the order of segments processing is not important, the approximation criterion giving the possibility to split them in the order of their appearance as it is happened in the recursive implementation. In the bounded-$\#$ approach this processing manner is not possible. In this case, to obtain a good approximation of the input curve, which means to have the smallest approximation error for the desired number of points, the segments must be processed in the decreasing order of their approximation error. Thus, due to this requirement and also because the recursive implementation does not offer the possibility to process the segments in the correct order, the only technique that can be applied to implement the bounded-$\#$ approach of the Recursive Split method is the iterative one.

The iterative implementation of the Recursive Split method in bounded-$\#$ approach is more complicated than the previous explained one. This is because in this new implementation the segments must be ordered regarding their approximation error. To process always the segment with the greatest approximation error a priority queue data structure was added. Thus, the correct processing order of the segments, which is necessary in obtaining the best solution of the bounded-$\#$ approach, is established by the following two queue operations:

- **Extraction** from the queue of the first element gives in each step the segment with the greatest approximating error
- **Insertion** in the queue of the two segments that result in a splitting operation must be done on the corresponding positions to preserve the elements decreasing order relative to the approximation error.

The iterative implementation of the bounded-$\#$ uses also a list data structure to store the segments of the approximation polygon. This list is similarly organized like that described above in the bounded-$\epsilon$ approach implementation.
The basic element of the priority queue stores the following information:

- **seg** is a link to the segment element from the polygonal approximation list
- **error** represents the approximation error of the current the segment
- **next** field used to link the elements of the priority list

The iterative implementation of the Recursive Split polygonal approximation in bounded-# approach is described below:

```plaintext
Recursive Split Algorithm, bounded-# approach, (RecSplit_{bn})

RecursiveSplit_{bn}(p_i, p_j, M) // compute the polygonal approximation
// of the curve p_i-p_j using M points
{
    // initiate the list with the first approximation segment
    // that links the two end points of the input curve
    list_elem:= new_element() // generate a new element of the list
    list_elem.next:=NULL
    list_elem.begin:=p_i
    list_elem.end:=p_j

    d:=error(p_i, p_j, &p_split) // the function error returns the error of
    // approximating the curve p_i-p_j, with the segment p_i-p_j, the splitting point p_split
    list_elem.split_pt:=p_split

    // initiate the priority queue with the first segment of the list
    priorityQ.seg:=list_elem
    priorityQ.error:=d
    priorityQ.next:=NULL
    n_points:=2

    while n_points < M do
        current_elem:= extract_priorityQ() // extract the first
        // element of the priority queue
        elem_new:=new_element() // generate a new element of the list
        elem_new.next:=current_elem.next
        current_elem.next:=elem_new
        new_elem.end:=current_elem.end
        new_elem.begin:=current_elem.split_pt
        current_elem.end:=current_elem.split_pt
        d:=error(current_elem.begin, current_elem.end, &p_split)
        current_elem.split_pt:=p_split
        insert_priorityQ(current_elem, d)
        d:=error(new_elem.begin, new_elem.end, &p_split)
        new_elem.split_pt:=p_split
        insert_priorityQ(new_elem, d)
        n_points ++
}
```
A common problem of all possible implementations of the Recursive Split method is the finding of the splitting point. In the classical manner of computation, the algorithm deals with the furthest curve point to the approximating segment (figure 19). In this case, finding the splitting point means to compute the maximum value of the distances between the curve points and the approximating segment.

![Fig. 19 Finding the splitting point as the furthest curve point to the segment](image)

A new approach used in finding the splitting point was introduced in this study. The alternative is to detect the curve point that divides the area between the curve and the approximating segment in two equal parts (figure 20). To find this point, the sum of the distances from the curve points to the segment must be computed.

In the both approaches, different kinds of distances can be used to determine the splitting point. Therefore, the entire software implementing the Recursive Split algorithm was designed in a generic manner. Thus, at the Recursive Split function could be plugged in different kind of distance traits classes.

![Fig. 20 The splitting point is dividing the area in two equal parts](image)
Using the dividing area method to detect the splitting point is not equivalent to computing the polygonal approximation in bounded-ε approach, based on the sum of distances criterion. In the above-explained approach, the sum of distances is used local only to split the area between the current curve portion and its approximating segment in two equal parts, but not as a global measure over the entire curve approximation.

To obtain an implementation of the recursive Split method dealing with the sum of distances criterion, the algorithm must cumulate the local errors in such a manner to be able to assess the sum of the distances over the entire input curve. The modification of the algorithm consist of adding the following two steps used to calculate the global approximation error:

- **subtract** the approximation error corresponding to the current segment that disappear from the polygonal approximation after the splitting operation
- **add** the new approximation errors characterizing the two segments obtained in the splitting process of the current segment.

Based on these modifications, the Recursive Split method using the sum of distances for error assessment is implemented as follows:

```plaintext
Recursive Split Algorithm, bounded-ε approach, (RecSplit_be_sum)
the error is assessed as the Sum of distances

RecursiveSplit_be_sum(p_i, p_n, epsi) // compute the polygonal approximation
  // of the curve p_i...p_n with an error=eps, sum of distances
  // less than or equal to the threshold epsi
{
  // initiate the list with the first approximation segment
  // that links the two end points of the input curve
  list_elem := new_element() // generate a new element of the list
  list_elem.next := NULL
  list_elem.begin := p_i
  list_elem.end := p_n

  d := error(p_i, p_n, &p_split) // the function error returns the error of
  // approximating the curve p_i...p_n, with the segment p_i...p_n
  // and also return the splitting point p_split

  list_elem.split_pt := p_split

  // initiate the priority queue with the first segment of the list
  priorityQ.seg := list_elem
  priorityQ.error := d
  priorityQ.next := NULL
  global_error := d

  while global_error < epsi do
    current_elem := extract_priorityQ() // extract the first
    // element of the priority queue
    global_error := global_error - error(current_elem)

    elem_new := new_element() // generate a new element of the list
    elem_new.next := current_elem.next
    current_elem.next := elem_new
    new_elem.end := current_elem.end
    new_elem.begin := current_elem.split_pt
```

33
4.3 Graph Search Method Implementation

To find an optimal solution to a polygonal approximation problem, the Graph Search method applies searching techniques to find the shortest path in the distances graph associated to the input curve. Applying the breadth-first search method, false loops in the resulting polygonal approximation can appear. Therefore, the breadth-first search technique was changed to process the points only in the forward direction, from the first point \( p_1 \) to the end point \( p_n \).

To obtain better results in running time, the reduced distances graph containing only the arcs with the weight less than or equal to the input error threshold \( \varepsilon \) is used. Also, for the same reason, in the construction of the distances graph, the on-line convex hull based Euclidean distance is applied.

To find the shortest path in the distances graph, the Graph Search method uses three auxiliary data structures:

- \( \text{visited} \) is a vector used to mark all the points that were already processed in the current step;
- \( \text{queue} \) is a queue structure used in ordering the points in the breath-first search manner;
- \( \text{poly_list} \) is a list structure storing the output approximation polygon.

The algorithm of the Graph Search method using the modified breadth search algorithm is:

```c
GraphSearch_be_sum(p1, pN, epsi) // compute the polygonal approximation // of the curve p_1-p_N with an error <= epsi
{
    // Distances graph construction
    for m:=2 to N do
        for n:=m+1 to N do
            Dist_Graph(n,m) := on_line_convex_hull_distance(n, m)
}```
// Searching the optimum polygonal approximation
// initiate the data structures
initiateQ(queue)
initiateL(poly_list)
initiateV(visited)
search_flag:=1
addQ(queue, p)
mark_visited(visited, p)
while search_flag=1 do
    p:=get_topQ(queue)
    for p:=p_l to p_h do // process the points only in the forward direction
        if Dist_Graph[p]<ε then // process only the arcs from
            if p=p_h then // the end of the curve is reached
                search_flag:=0
                break
        if not visited(visited, p) then
            add_list(poly_list, p)
            mark_visited(visited, p)
            addQ(queue, p)
        }

### 4.4 Distance Traits Classes Implementation

One of the advantages of using generic programming techniques to implement polygonal approximation algorithms consists of the possibility to preserve the same function code and to plug in different kind of distance traits classes. Thus, without any change in the function implementation, several polygonal approximations with different kind of error assessment can be easily obtained.

When the software is designed in a generic manner, important features like compactness, flexibility and generality are conferred to the resulted code.

#### 4.4.1 Implementation Solutions

In this section are pointed out the steps and the decisions made in the strategy of implementing the distance traits classes that are based on distances from \( L_p \) category. To illustrate the progress obtained in software development, the algorithms presented here refer only to the particular case of the Euclidean distance measured to the line support. Similar improvement steps were applied also for other distances from the \( L \) category, like: Manhattan distance and \( L_\infty \) distance measured either to the segment or to the line support. Moreover, the same strategies were used in developing the software for vertical distance. The experimental results obtained for all these distances are similar to those presented here for the Euclidean distance.
To assess the error obtained in approximation of a curve by the segment linking its end points, the distances between the points of the curve and the segment must be computed. Therefore, the initial idea used in designing the distance traits classes software was to solve the basic geometric operations by applying the corresponding functions from the CGAL library, which were already available. Thus, the first implementation was based on the following algorithm:

**Euclidean Distance Algorithm using CGAL-functions (EuclDistCGAL)**

```plaintext
MaxEuclideanDistance(p_1, p_n) // compute the approximation error
    // between the curve p_1 to p_n and the segment p_n0
{
    p_c:=p_1 // initialize the current point
    max:=0
    p_max:=p_1
    do
        p_c:=+ // take the next point
        d:=CGAL_EuclideanDistance(p_c, p_p) // compute the distance
            // from the point p_i to the segment p_p0
        if d > max then
            max:=d
            p_max:=p_c
    while p_c != p_n
}
```

Unfortunately, poor experimental performances were obtained for this version of computing the approximation error. The first step made to improve the running time of the algorithm was to replace the calling of the CGAL functions with equivalent own-implemented *inline* code. Putting all the code inside the distance function, the running time decreases because of eliminating the time needed before to call the CGAL functions and therefore is not expected a great improvement of the performances. With these modifications, the new algorithm is:

**Euclidean Distance Algorithm using inline code (EuclDistInline)**

```plaintext
MaxEuclideanDistance(p_1, p_n) // compute the approximation error
    // between the curve p_1 to p_n and the segment p_n0
{
    p_c:=p_1 // initialize the current point
    max:=0
    p_max:=p_1
    do
        p_c:=+ // take the next point
        // compute inline the distance from the point p_i to the segment p_p0
        d_c_x:=p_c.x - p_i.x
        d_c_y:=p_c.y - p_i.y
        d:=((p_c.y-p_i.y)*d_c_x - (p_c.x-p_i.x)*d_c_y)
        d:=d*d / (d_c_x^2 + d_c_y^2)
```
if $d > \text{max}$ then
    \text{max} = d
    \text{p}_\text{max} = \text{p}_c$
    \text{while} \ p_c \neq \text{p}_n$

Further improvement of the algorithm used to assess the approximation error is based on an observation that will be discussed in the following. The error introduced by replacing a curve portion with the segment linking its end points is measured by computing all the distances between the points of the curve and the approximation segment. During this procedure, the parameters of the approximation segment remain unchanged and the current point is moving from the beginning to the end point of the curve. So, it is easy to observe that in computing the approximation error there are used two categories of parameters: those with constant value, which are related to the approximation segment features and the parameters with variable value that depend on the coordinates of the current point changing along the curve. According to this observation, the algorithm computing the approximation error can be improved by splitting the previous version in two parts:

- Computation of the constant parameters that are not related to the current point, which is done at the beginning of the procedure, outside the loop used to move the points along the curve;
- Computation of the parameters depending on the coordinates of the current point that is done inside the loop crossing the curve, because for each step of the loop the coordinates of the current point change and the parameters must be recomputed.

After these specifications, the new improved algorithm used to compute the approximation error is:

**Euclidean Distance Algorithm Improved Version (EuclDistImp)**

\[
\text{MaxEuclideanDistance}(\text{p}_1, \text{p}_n) \quad \text{// compute the approximation error}
\quad \text{// between the curve p_1p_n and the segment p_1p_c}
\{
    \text{p}_c := \text{p}_1 \quad \text{// initialize the current point}
    \text{max} := 0
    \text{p}_\text{max} := \text{p}_1

    \text{// Compute the parameters depending on the segment p_1p_c features}
    \text{c}_x := \text{p}_1.x - \text{p}_1.x
    \text{c}_y := \text{p}_1.y - \text{p}_1.y
    \text{d}_c := \text{c}_x^2 + \text{c}_y^2
    \text{v} := \text{p}_1.x \cdot \text{p}_1.y + \text{p}_1.y \cdot \text{p}_1.x

    \text{do}
    \quad \text{p}_c := \text{p}_c + \quad \text{// take the next point}
    \text{// Compute the local parameters depending on the current point p_c coordinates}
    \text{t} := v - \text{p}_c.x \cdot dy + \text{p}_c.y \cdot dx
    \text{d}_t := (\text{p}_c.y - \text{p}_1.y) \cdot \text{c}_y - (\text{p}_c.x - \text{p}_1.x) \cdot \text{c}_x
    \text{d}_c := t^2 / d

    \quad \text{if} \ d > \text{max} \ \text{then}
    \quad \text{max} := d
    \quad \text{p}_\text{max} := \text{p}_c

    \text{while} \ p_c \neq \text{p}_n
    \}
\]
To compare the performances of the distance assessing algorithms discussed above, in table 3 are presented the experimental values of running time obtained for Dynamic Programming polygonal approximation method, in bounded-# (min-c) approach. These are the experimental data obtained in the particular case of approximating the curve from file "kk1.dat" (figure 21), with \( N=560 \) input number of points and \( M=64 \) desired output number of points.

![Figure 21](image)

**Fig. 21** Example of input curve from set "fish"; file "kk1.dat".

The experiments made for different polygonal approximation methods (Dynamic Programming, Recursive Split, in both bounded-# and bounded-c approaches) and for different distances used to assess the approximation error (Euclidean, Manhattan, \( L_{\infty} \), vertical) show similar ratios between the three versions of implementation like those given in table 3.

<table>
<thead>
<tr>
<th>Maximum Euclidean Distance Algorithm</th>
<th>Distance measured to the line support</th>
<th>Distance measured to the segment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time [s]</td>
<td>ratio</td>
</tr>
<tr>
<td>EuclDistCGAL</td>
<td>374.137</td>
<td>1.000</td>
</tr>
<tr>
<td>EuclDistInline</td>
<td>141.814</td>
<td>0.379</td>
</tr>
<tr>
<td>EuclDistImp</td>
<td>36.392</td>
<td>0.097</td>
</tr>
</tbody>
</table>

Table 3 Comparison of the experimental running time measured for the different implementation of the Euclidean distance. The ratios are computed taking as reference the time obtained for EuclDistCGAL version.
4.4.2 Exact Computation

Among the advantages of using generic techniques in implementing the polygonal approximation algorithms is the possibility to plug in and, thus, to make computations in different kind of kernel representations like Cartesian and Homogeneous. The Homogeneous kernel uses rational numeric representation that offer the possibility to evaluate all the calculations in exact arithmetic and thus to obtain an exact value of the final result of the processed algorithm. Therefore, according to this idea, one of the requirements that had to be accomplished in designing the distance assessment algorithms was to find a computation method capable to provide the exact value of the distance. To this purpose all the operations that are evaluated by the computer using approximation techniques, like: square root (SQRT) and all the trigonometric operations (sin, cos), had to be avoided. This restriction imposed to the manner of implementation imply one of the following two cases:

- There are algorithms that can be solved using either inexact computation or exact computation strategies, but usually the former method is faster that the latter one.

- There are algorithms that admit only the solution implying operations evaluated by inexact computation. In these cases, an upper bound value computed in exact arithmetic, which is a good approximation of the real solution, must be found.

4.4.2.1 The Influence in Running Time

As explained before, an implementation of the polygonal approximation algorithms using only operations involving exact computation was required. To observe the influence in running time of such a strategy of implementation, several methods, both from the exact computation and the inexact computation categories, were studied.

This section points out the difference in running time between the two categories of algorithms. The EuclDistImp algorithm, which is given in section 4.2.1, represents the best implementation of the Euclidean distance using only exact computation operations. For the second category, a good implementation of the Euclidean distance assessment using trigonometric functions, which implies inexact calculations, is presented next:

```plaintext
Euclidean Distance Algorithm Trigonometric Version (EuclDistTrig)

MaxEuclideanDistance(p1, p2) // compute the approximation error
    // between the curve p1p2 and the segment p2
{
    p0 := p1; //initialize the current point
    max := 0
    pmax := p1

    // compute the parameters depending on the segment p1p2, features
    alpha := -atan((p2.x - p1.x) / (p2.y - p1.y))
```

39
do
    $p_c+$ // take the next point
    $d:=p_c.y \cdot \cos(\text{alfa}) + p_c.x \cdot \sin(\text{alfa})$
    if $d > \text{max}$ then
        $\text{max}:=d$
        $p_{\text{max}}:=p_c$
    end if
while $p_c \neq p_n$

The comparison of the running times presented in figure 22 is obtained in an experiment of polygonal approximation based on Dynamic Programming method, bounded # approach.

The approximation error is assessed using the maximum Euclidean distance, which is measured either to the line support or to the segment, is implemented in both manners: using exact computation operations and using trigonometric functions (inexact computation). The average value of the ratio between the running time of the inexact method and the running time of the exact method is 0.456 for the distances measured to the line support and 0.482 for the distances measured to the segment.

The entire set of the data measured in this experiment is given in table 7 in annex, page 59.

Fig.22 Running time experimental results comparison between the exact computation version (Exact) and the trigonometric based (Inexact) implementations of the Euclidean Distances assessment algorithm.
4.4.2.2 Finding Upper Bound of the Real Value

Sometimes the entities that are required to be evaluated cannot be computed avoiding operations like square root or trigonometric functions. In these cases, the only solution of the exact computation is to find an upper bound function that approximate the real value and can be assessed using only exact computation operations.

This is the situation appeared in evaluating the bounding shell distance. The bounding shell distance between a point and a segment is defined as the maximum distance between the segment and the arc passing through the end points of the segment and the input point. If the length of the segment is $L$ and the radius of the circle containing the arc is $R$ (figure 23), then the bounding shell distance is:

$$d_{BS} = R - \sqrt{R^2 - \frac{L^2}{4}}$$

This value cannot be computed without using the square root operation. Therefore, the only way to use this distance in exact computing applications is to find an upper bound function that is a good approximation of the real distance and that can be computed in exact arithmetic.

The best upper bound function found to approximate the bounding shell distance is:

$$d_{BS-\text{up}}^2 = \frac{\left(\frac{L}{2}\right)^4}{4R^2 - 3\left(\frac{L}{2}\right)^2}$$

To point out the approximation quality of the bounding shell distance, in figure 24 is presented the variation of the ratio between the upper bound function and the real value. As it can be observed from the figure, if the radius $R$ is greater than $2L$, then the approximation error is less the 3%.

Fig.23 The elements used in computing the bounding shell distance.

41
4.4.2.3 Implications of the Exact Computation

Due to the generic manner of software designing, different kinds of kernels and base data types can be used to solve polygonal approximation tasks. As a direct result of this facility, there exists the possibility to calculate the correct solution, without rounding errors, of the polygonal approximation problems using only exact computation operations. To be more specific, the exact computation applications can be obtained in two ways related to the manner of combining the kernel representations and the base data types:

- Using the Cartesian kernel based on an exact real number format like the `leda_real` type;
- Using the Homogeneous kernel, which implies rational number representation, based on exact integer number format like: `leda_integer` and `GmpZ`.

Regardless the kernel type, in an exact computation task the base data type must always provide the correct (errorless) number value, without any alteration of it. This requirement is not easy to be accomplished, because of the following restrictions that characterized, at least, the standard number types:

- The rounding operations that are usually applied in the real number representations
- The bounded range of the variables

Both of these two inconveniences are the result of the fixed memory space allocation for each type of variable. For the Homogeneous kernel, when the rational number representation is used, all the operations are basically calculated by multiplication between integer values. In such a situation the
values of the numerators and the denominators are increasing exponentially. This conclusion was practically confirmed in an experiment using the Homogeneous kernel based on the long int base data type, in which the correct solution of the polygonal approximation can be obtained for input curves with less than 30 points. For more complex input curves the numbers became greater then the upper bound of the long int number type range of representation and the final result is a wrong solution of the approximation problem.

This manner of constant memory space allocation used in the standard memory management is not possible to be applied in designing the exact computation applications that don't admit rounding operations and deal with values in different ranges of variation. The solution of this problem consists of using special data types using adaptive dynamic memory space allocation, capable to store any value without loss of precision. This solution is generally valid from a theoretical point of view, because for practical reasons it can be applied only in the limit of the available computer memory. Moreover, the time needed to process such an adaptive number structure is increasing with the dimension of the memory space allocated to it.

So, two negative effects appeared in processing the exact computation tasks based on rational number representation: the exponential increase of the memory space allocated to the new data types that implies also the increase of the time needed to process them. The solution applied to improve these problems was a rescaling technique. To decrease as much as possible the memory space the rescaling method eliminates all the common factors of the numerator and the denominator for each rational number. This idea is implemented by a function computing the greatest common factor that is called after each operation involving rational numbers.

The experimental running times measured for Dynamic Programming polygonal approximation method, bounded-# approach, computed using different kinds of kernels and base data types are graphical represented in figure 25. More details about data and the measuring conditions are given in annex, table 8, page 60.

---

**Fig. 25** The running time experimental results measured for implementations of the Dynamic Programming method using different kind of kernels and base data types.
As can be observed from the graphic, for a small number of input points, implying a reduced number of operations, the implementation using the rescaling technique is worse than the direct rational number computation. The reason of this behavior is the space used to store the rational numbers that is still small enough and therefore the time needed for adjusting operations is greater than the time used to process directly the rational numbers. But for greater numbers of input points, the number of operations increases and a larger space is allocated for the exact number representations. Thus, in this case the rescaling technique becomes inefficient, the time spent for number-adjusting operations is smaller than those used in the direct processing implementation.

Another conclusion observed from the experimental measurements is the similar behaviors of the implementation using the Cartesian kernel based on the leda_real number type and the case of applying the Homogeneous kernel with the leda_integer and rescaling technique.

### 4.4.3 Notes About the Path Hull Distance Implementation

An attempt to improve the computation of the maximum Euclidean distance between a curve portion and the segment linking its ends is to decrease the number of points that are considered valid in computing it. In the classical implementation all the points of the curve portion are used to find the maximum Euclidean distance, but not all of them are good candidates of the furthest point to the segment. Thus, an improvement of the algorithm can be obtained by eliminating all the points that are already known as bad solutions of the furthest point to the segment. This is possible by computing the convex hull of the curve portion, because all the points that are not part of the convex hull are inside of it and therefore they are at a smaller distance to the segment than are the points belonging to the convex hull.

The on-line method based on the path hull data storing structure is applied to compute the convex hull of the curves. When this technique is applied in a Recursive Split polygonal approximation task, in each step the furthest point divides the current curve in two sub-curves and corresponding to this operation, the current hull must be split also in two sub-hulls related to the two sub-curves. To accomplish this operation two methods were implemented.

The first method consists of introducing a history stack structure that is used to restore the previous stages of building the current hull. In such a way, one of the subhulls resulting from the splitting operation can be recovered from the current hull by calling back from the history stack the previous operations used to construct the current hull. The second sub-hull must be build from the beginning using the on-line convex hull algorithm.

The alternative of this strategy is to build always both of the sub-hulls. In this kind of implementation the history stack is not needed and also the function recovering the previous stages of the current hull is useless. Actually in this case only the on-line convex hull algorithm is applied twice: to the set of curve points that are before the current furthest point and to set of curve points that are after it.

Studying these two implementations from a practical point of view, interesting conclusions were found. In table 4 are presented the results obtained running the two versions of the path hull distance.
Table 4: Running time comparison between the restore and the build versions of path hull distance implementation. The results are given for two categories of curves: with large convex parts (kk1.dat) and without large convex parts (random generated).

<table>
<thead>
<tr>
<th>curve</th>
<th>N</th>
<th>M</th>
<th>Time [s]</th>
<th>Restore</th>
<th>Build</th>
<th>Build time</th>
</tr>
</thead>
<tbody>
<tr>
<td>kk1.dat</td>
<td>560</td>
<td>79</td>
<td>0.600</td>
<td>0.100</td>
<td>6.000</td>
<td></td>
</tr>
<tr>
<td>kk1.dat</td>
<td>760</td>
<td>100</td>
<td>0.800</td>
<td>0.141</td>
<td>5.674</td>
<td></td>
</tr>
<tr>
<td>random</td>
<td>360</td>
<td>211</td>
<td>0.080</td>
<td>0.131</td>
<td>0.611</td>
<td></td>
</tr>
<tr>
<td>random</td>
<td>760</td>
<td>352</td>
<td>0.151</td>
<td>0.511</td>
<td>0.296</td>
<td></td>
</tr>
</tbody>
</table>

For the curves that contain large convex parts the recovering operation became expensive in running time. This is because the convex hull is almost similar with the processed curve portion and the operations needed to manage the history stack are time consuming. Therefore, in these cases the second solution based on building the both sub-hulls gives better results.

On the other hand, if the input curves have frequent changes of direction that interrupts its convexity then the recovering version gives better results. In these cases the building of the convex hull involves more operations than is needed to recover it from the previous one.

The experimental results given in table 4 are obtained using the curves presented in figure 21 (page 37) and figure 26. The first curve, stored in file “kk1.dat”, corresponds to the first category that contains large convex parts. The second curve with small convex parts was random generated.

Fig. 26: Example of artificial generated curve, containing small convex parts.
### 4.4.4 Two-Dimensional and Three-Dimensional Distance Implementations

For more generality of the polygonal approximation software, a library containing 3-dimensional distance algorithms was added to it. Thus, a new feature is available: by using the generic implemented polygonal approximation functions together with the 3-dimensional distance traits classes, now curves from a 3-dimensional space can be processed.

The 3-dimensional distance algorithms were designed using the experience cumulated during the implementation and optimization of the corresponding 2-dimensional methods. Similar techniques with those used in 2-dimensional case, like computing the current point depending parameters inside the curve crossing loop and the others before the loop, were adapted to the 3-dimensional applications. To exemplify the correlation between the 2-dimensional and the 3-dimensional methods, the 3-dimentional version of the algorithm used to assess the maximum Euclidean distance measured to the line support that is given below can be compared with its 2-dimensional version (EuclDistImp) presented in section 4.2.1.

**Euclidean Distance Algorithm 3D Version (EuclDist3D)**

```plaintext
MaxEuclideanDistance_3D(p, pN)  // compute the approximation error
// between the curve p...pN and the segment pN
{
    pc := pN  //initialize the current point
    max := 0
    pMmax := pN
    // compute the parameters depending on the segment pN-p
    dX := pN.x - p.x
    dY := pN.y - p.y
    dZ := pN.z - p.z
    dk := dX * dx + dY * dy + dZ * dz
    do
        pc++  // take the next point
        // Compute the local parameters depending on the current point pc coordinates
        deltaX := p.x - pc.x
        deltaY := p.y - pc.y
        deltaZ := p.z - pc.z
        t := (deltaX * dx + deltaY * dy + deltaZ * dz) / dk
        cx := deltaX - dX * t
        cy := deltaY - dY * t
        cz := deltaZ - dZ * t
        d := cx^2 + cy^2 + cz^2
        if d > max then
            max := d
            pMmax := pc
        while pc != pN
    }
```

46
The results of running time obtained for the 2-dimenSional and the 3-dimenSional implementations of polygonal approximation methods based on different kind of distance assessments are given in table 5 and also some of them are graphical represented in figure 27.

More exactly, in figure 27 are represented the variations of the running time obtained for Dynamic Programming method, bounded-# approach, using different distances measured to the line support, applied for solving polygonal approximation applications in 2-dimensional and 3-dimensional spaces. The data used in this representation are given in table 9 from annex, page 60.

![Figure 27](image)

**Fig. 27** The running time experimental results' comparison between different kind of distances measured for 2-dimensional and 3-dimensional implementations.

In table 5 are presented the average value of the ratios between the running times obtained for the 2-dimensional and the 3-dimensional implementations. This time are given the results of a more exhaustive study, the measurements covering almost all the possible polygonal approximation applications:

- using the Dynamic Programming and the Recursive Split methods,
- in both bounded-# and bounded-ε approaches,
- assessing the error as the maximum distance or as the sum of the distances,
- the distances are measured either to the line support or to the segment
- the input curves are in 2-dimensional or in 3-dimensional space.
As was expected the 3-dimensional applications run slower than the 2-dimensional implementations. For the distances measured to the line support they are about 3 times slower and for the distances measured to the segment they become approximately 2 times slower. This difference is because the 2-dimensional distances measured to the line support are easier to compute.

<table>
<thead>
<tr>
<th>Distance</th>
<th>Average ratio of 2D / 3D running times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bounded-$e$ (e=1)</td>
</tr>
<tr>
<td>MaxEuclid_Ls</td>
<td>0.331</td>
</tr>
<tr>
<td>MaxInfinity_Ls</td>
<td>0.327</td>
</tr>
<tr>
<td>MaxManhattan_Ls</td>
<td>0.334</td>
</tr>
<tr>
<td>MaxEuclid_seg</td>
<td>0.463</td>
</tr>
<tr>
<td>MaxInfinity_seg</td>
<td>0.488</td>
</tr>
<tr>
<td>MaxManhattan_seg</td>
<td>0.497</td>
</tr>
<tr>
<td>SumEuclid_Ls</td>
<td>0.343</td>
</tr>
<tr>
<td>SumInfinity_Ls</td>
<td>0.457</td>
</tr>
<tr>
<td>SumManhattan_Ls</td>
<td>0.473</td>
</tr>
<tr>
<td>SumEuclid_seg</td>
<td>0.540</td>
</tr>
<tr>
<td>SumInfinity_seg</td>
<td>0.653</td>
</tr>
<tr>
<td>SumManhattan_seg</td>
<td>0.565</td>
</tr>
</tbody>
</table>

4.5 Comparison Between Polygonal Approximation Methods

Besides the theoretical research involved in this work, there existed a practical purpose of it consisted of designing a software library containing polygonal approximation functions and their auxiliary distance traits classes. To reach this goal, a comprehensive analysis covering different categories of algorithms and implementation techniques was performed. Programming techniques like generic and dedicated implementations, recursive and iterative schemes were applied to determine their benefits in the practical behavior of the algorithms. Also, different methods from optimal and sub-optimal classes were studied and tested to find the correspondence between the loss of the solution optimality and the practical running time improvement. The results and the conclusions obtained in all these experiments are discussed below classified relative to the polygonal approximation approach and to the distance type used to assess the approximation error.
4.5.1 Bounded-ε Approach, Sum of the Euclidean Distances

In this case three algorithms were studied:

- *Dynamic Programming* from optimal category, implemented in generic manner.
- *Recursive Split* from sub-optimal class of algorithms, generic implemented.
- *Graph Search*, in two versions:
  - sub-optimal, using the breadth first search algorithm;
  - optimal, based on finding the shortest path in a weighted graph algorithm [9].

Both versions are implemented in a dedicated way, using only the sum of the Euclidean distances to assess the approximation error.

The running time experimental results are presented in figure 28. As was expected, the Dynamic Programming is the slowest algorithm and the Recursive Split is the fastest one. It is interesting to note the improvement of the running time performances when the Dynamic Programming implementation is running using the incremental technique to assess the sum of the Euclidean distances.

The entire set of experimental data and the measuring conditions are given in annex, table 10, page 61.

![Graph showing performance comparison](image.png)

**Fig. 28** Comparison of the running time experimental results obtained for different polygonal approximation algorithms for the bounded-ε approach, using the sum of the Euclidean distances as error assessment.
In figure 29 are graphically represented the variations of the output number of points that results applying different polygonal approximation algorithms. Compensating the gain obtained in running time, the variation of the output number of points obtained for the sub-optimum algorithms presents a deviation from the optimum values. As it can be observed, the performances in approaching the optimum values of the output number of points are in reverse order relative to the running time order: the faster is the implementation, the greater is the deviation from the optimum output results.

![Graph](image)

**Fig. 29** Variation of the output number of points comparison between the results obtained for optimal and sub-optimal polygonal approximation algorithms in bounded approach, using the sum of the Euclidean distances as error assessment.

Figure 30 presents the comparison between the running times obtained for Dynamic Programming using incremental distance computation and the optimal Graph Search implementation. As it was expected, the results are similar, because in the above-mentioned conditions the two algorithms become equivalent: optimal algorithms using the sum of the Euclidean distance error assessment. The small variation between the experimental running time results comes from the implementation difference that exists between the two algorithms: the Dynamic Programming uses the generic techniques and the Graph Search is implemented in a dedicated way.
4.5.2 Bounded-$\varepsilon$ Approach, Maximum of the Euclidean Distances

Three algorithms were studied:

- *Dynamic Programming* from optimal category, implemented in generic manner.
- *Recursive Split* from sub-optimal class of algorithms, generic implemented.
- *Graph Search*, which is based on the breadth first search algorithm, finds in this approach an optimal solution to a polygonal approximation problem. Applying only the maximum of the Euclidean distances to assess the approximation error, a dedicated scheme was chosen to implement this algorithm.

Two parameters were analyzed and compared: the running time and the resulting number of points obtained in the polygonal approximation.

The running time experimental results are presented in figure 31. The Dynamic Programming is the slowest algorithm, the Recursive Split is the fastest one and the Graph Search has medium results. Maybe it is interesting to point out the Graph Search running time results that are better than the Dynamic Programming even if both of them are optimal methods. But, on the other hand, this conclusion is compensated by the generality and the flexibility offered by the Dynamic Programming algorithm that can be applied using different kind of distances and different distance cumulating techniques (maximum and sum) used to assess the approximation error, relative to the rigidity of the Graph Search method that finds an optimal solution only for the maximum Euclidean distance error assessment. Moreover, as it was already presented in section 4.5.1, the Graph Search method based on the breadth first search algorithm becomes sub-optimal when the sum of the Euclidean distances is used to compute the approximation error.
The variations of the output number of points that results applying different polygonal approximation algorithms are presented in figure 32. As it was expected, there is a deviation from the optimum solutions of the results obtained applying the sub-optimal method (Recursive Split).

More details about the experimental data and the measuring conditions are given in annex, table 11, page 62.
4.5.3 Bounded\# Approach, Sum of the Euclidean Distances

The algorithms studied in this case are:

- *Dynamic Programming* from optimal category, implemented in generic manner.
- *Recursive Split* from sub-optimal class of algorithms, generic implemented,
- *Graph Search*, optimal algorithm based on finding the shortest path in a weighted graph algorithm [9]. The implementation is dedicated only to the sum of the Euclidean distances approximation error assessment.

Similar results and conclusions with those presented in section 4.4.1 are obtained also for this case.

With respect to the running time performances the algorithms are ordered from the fastest that is the Recursive Split method, followed by the Graph Search algorithm, ending with the Dynamic Programming method that gives the poorest results for the classical implementation. A good improvement of Dynamic Programming time results is obtained when in error assessment the incremental technique is applied to compute the sum of the Euclidean distances. All these results are given in graphical form in figure 33.

![Graph](image)

**Fig. 33** Comparison of the running time experimental results obtained for different polygonal approximation algorithms for the bounded\# approach, using the sum of the Euclidean distances as error assessment.
Fig. 34 Variation of the approximation error: comparison between the results obtained for optimal and sub-optimal polygonal approximation algorithm in bounded-# approach, using the sum of the Euclidean distances as error assessment.

The advantage of the sub-optimal method is the good rate of its processing time, but this efficiency in time performance is paid by the loss of precision in finding the correct solution. In this approach the correctness of the solution is measured by the approximation error obtained for a desired output number of points. This error is minimal for the optimal methods (Dynamic Programming, Graph Search) and it has greater values for the sub-optimal methods. As a confirmation of these comments, in figure 34 are presented the variations of the approximation error obtained for optimal and sub-optimal methods.

The entire set of experimental data and the measuring conditions are given in annex, table 12, page 63.

4.5.4 Bounded-# Approach, Maximum of the Euclidean Distances

Three algorithms were studied:

- Dynamic Programming from optimal category, implemented in generic manner.
- Recursive Split from sub-optimal class of algorithms, generic implemented.
- Graph Search uses the breadth first search algorithm to find an optimal solution to a polygonal approximation problem. The dedicated implementation uses only the maximum of the Euclidean distances to assess the approximation error.

The running time experimental results are presented in figure 35. Similar with the results obtained in the previous experiments, also in this one the best running time results are obtained for the Recursive Split implementation and opposite to it is the Dynamic Programming method.
In figure 36 are given the polygonal approximation errors obtained for different algorithms. The Dynamic Programming and the Graph Search methods give for each case the minimum values of the approximation error that is the optimum solution at the bounded-# problem. On the contrary, the Recursive Split algorithm finds worse solutions, implying errors greater than the minimum possible values.

More details regarding the experimental data and the measuring conditions are given in annex, table 13, page 64.
Annex

All the experimental measurements presented in this study were obtained using a DELL computer with the following features:

- Intel Pentium III, 1GHz processor
- 512 kB RAM
- 20 GB HDD
- Windows 2000 operating system

For each table from this annex is given the conditions used to obtained the experimental results that are included in it. The experimental results presented here are obtained processing the input curve stored in file "dcl1.dat" (fig. 21, page 37), but during the testing stage, similar results and conclusions were obtained using different input curves.

Table 1  Experimental measurements of running time obtained for the old and the new implementations of Dynamic Programming method, bounded-e (min#) approach

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>M</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>44</th>
<th>64</th>
<th>96</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynProg-E-OLD</td>
<td>36.392</td>
<td>107.945</td>
<td>243.693</td>
<td>418.452</td>
<td>507.367</td>
<td>596.411</td>
<td>642.129</td>
<td>664.868</td>
<td></td>
</tr>
</tbody>
</table>

Table 2  The influence of the output number of points M in running time of new implementation of the Dynamic Programming method, bounded-e approach

<table>
<thead>
<tr>
<th>M</th>
<th>Error</th>
<th>Distance</th>
<th>Search</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>469,241129</td>
<td>91,111</td>
<td>0,171</td>
<td>91,282</td>
</tr>
<tr>
<td>10</td>
<td>35,945359</td>
<td>91,161</td>
<td>0,441</td>
<td>91,602</td>
</tr>
<tr>
<td>20</td>
<td>8,513787</td>
<td>91,121</td>
<td>0,982</td>
<td>92,103</td>
</tr>
<tr>
<td>30</td>
<td>3,723077</td>
<td>90,921</td>
<td>1,622</td>
<td>92,543</td>
</tr>
<tr>
<td>40</td>
<td>1,744813</td>
<td>91,101</td>
<td>2,534</td>
<td>93,635</td>
</tr>
<tr>
<td>50</td>
<td>1,212871</td>
<td>90,941</td>
<td>3,645</td>
<td>94,586</td>
</tr>
<tr>
<td>60</td>
<td>1,000000</td>
<td>91,234</td>
<td>4,836</td>
<td>96,07</td>
</tr>
<tr>
<td>70</td>
<td>0,941176</td>
<td>91,061</td>
<td>5,853</td>
<td>96,92</td>
</tr>
<tr>
<td>80</td>
<td>0,844828</td>
<td>91,041</td>
<td>6,941</td>
<td>97,982</td>
</tr>
<tr>
<td>90</td>
<td>0,752941</td>
<td>91,131</td>
<td>7,702</td>
<td>98,833</td>
</tr>
<tr>
<td>100</td>
<td>0,634518</td>
<td>91,982</td>
<td>8,762</td>
<td>100,744</td>
</tr>
<tr>
<td>200</td>
<td>0,243243</td>
<td>91,051</td>
<td>18,453</td>
<td>109,504</td>
</tr>
<tr>
<td>300</td>
<td>0,200000</td>
<td>91,031</td>
<td>28,256</td>
<td>119,287</td>
</tr>
</tbody>
</table>
Table 3: The influence of the input number of points $N$ in running time of new implementation of the Dynamic Programming method, bounded-$\epsilon$ approach.

**Experimental conditions:**
- Algorithm: Dynamic Programming, bounded-$\epsilon$ (min-#) approach
- Input curve: file "kkl.dat"
- Distance: Maximum Euclidean Distance, measured to the line support
- Output number of points $M=50$

<table>
<thead>
<tr>
<th>N</th>
<th>Distance [s]</th>
<th>Search [s]</th>
<th>Total [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.191</td>
<td>0.021</td>
<td>0.212</td>
</tr>
<tr>
<td>200</td>
<td>1.633</td>
<td>0.081</td>
<td>1.714</td>
</tr>
<tr>
<td>300</td>
<td>5.257</td>
<td>0.311</td>
<td>5.568</td>
</tr>
<tr>
<td>400</td>
<td>11.937</td>
<td>0.581</td>
<td>12.518</td>
</tr>
<tr>
<td>500</td>
<td>22.933</td>
<td>0.982</td>
<td>23.915</td>
</tr>
<tr>
<td>600</td>
<td>39.916</td>
<td>1.533</td>
<td>41.449</td>
</tr>
<tr>
<td>700</td>
<td>61.368</td>
<td>2.464</td>
<td>63.832</td>
</tr>
<tr>
<td>800</td>
<td>91.011</td>
<td>3.655</td>
<td>94.666</td>
</tr>
</tbody>
</table>

Table 4: The running time experimental results, comparison between the new (NEW) and the new-improved (NEW, IMP) implementations of the Dynamic Programming method, bounded-$\epsilon$ (min-#) approach.

**Experimental conditions:**
- Algorithm: Dynamic Programming, bounded-$\epsilon$ (min-#) approach
- Input curve: file "kkl.dat"

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>MaxEuclid_Dist_is</th>
<th>SumEuclid_Dist_is</th>
<th>Dymp_Bounded-#</th>
<th>Dymp_Bounded-#</th>
<th>Dymp_Bounded-#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>NEW_NEW_IMP</td>
<td>NEW_NEW_IMP</td>
<td>NEW_NEW_IMP</td>
<td>NEW_NEW_IMP</td>
<td>NEW_NEW_IMP</td>
</tr>
<tr>
<td>50</td>
<td>200</td>
<td>1.743 1.538 1.101</td>
<td>1.722 1.733 0.983</td>
<td>0.181 0.123 1.503</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>5.556 5.387 1.031</td>
<td>5.648 5.048 1.115</td>
<td>0.571 0.381 1.498</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>12.718 12.388 1.023</td>
<td>12.669 12.449 1.012</td>
<td>1.062 0.701 1.515</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>24.345 23.354 1.0424</td>
<td>24.245 23.426 1.032</td>
<td>1.723 1.152 1.495</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>41.039 39.647 1.0351</td>
<td>41.249 39.857 1.039</td>
<td>2.594 1.732 1.497</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>64.686 62.431 1.0413</td>
<td>64.593 62.411 1.035</td>
<td>3.906 2.344 1.566</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>96.001 91.988 1.0443</td>
<td>95.678 92.223 1.0396</td>
<td>5.648 2.966 1.885</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>200</td>
<td>1.893 1.689 1.1195</td>
<td>1.722 1.523 1.1307</td>
<td>0.201 0.140 1.435</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>6.159 5.328 1.1560</td>
<td>5.718 5.468 1.0457</td>
<td>0.731 0.501 1.459</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>13.951 12.369 1.1289</td>
<td>12.918 12.386 1.0428</td>
<td>1.461 0.901 1.520</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>45.676 39.927 1.1440</td>
<td>42.589 40.336 1.0558</td>
<td>4.136 1.933 2.073</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>70.741 62.501 1.1318</td>
<td>66.766 62.851 1.0623</td>
<td>6.109 2.754 2.218</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>105.269 92.493 1.1380</td>
<td>99.192 92.864 1.0681</td>
<td>8.292 3.716 2.234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>1.832 1.672 1.0957</td>
<td>1.773 1.542 1.1498</td>
<td>0.231 0.150 1.540</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>5.668 5.448 1.0404</td>
<td>5.678 5.318 1.0677</td>
<td>0.751 0.541 1.352</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>13.761 12.578 1.0941</td>
<td>13.711 12.668 1.0823</td>
<td>2.013 0.951 2.117</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>44.183 40.288 1.0967</td>
<td>44.013 40.366 1.0903</td>
<td>5.819 2.294 2.536</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>72.938 62.881 1.1596</td>
<td>68.759 63.351 1.0854</td>
<td>8.091 3.196 2.532</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>106.913 92.994 1.1497</td>
<td>100.985 93.368 1.0814</td>
<td>10.786 4.266 2.528</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

57
<table>
<thead>
<tr>
<th>N</th>
<th>MaxEuclidianDist</th>
<th>Error</th>
<th>DynProg</th>
<th>Bounded - ε</th>
<th>NEW</th>
<th>NEW_IMP</th>
<th>NEW</th>
<th>NEW_IMP</th>
<th>DynProg</th>
<th>Bounded - ε</th>
<th>NEW</th>
<th>NEW_IMP</th>
<th>DynProg</th>
<th>Bounded - ε</th>
<th>NEW</th>
<th>NEW_IMP</th>
<th>DynProg</th>
<th>Incremental Implementation</th>
<th>NEW</th>
<th>NEW_IMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.1</td>
<td></td>
<td>0.211</td>
<td>0.210</td>
<td>1.0048</td>
<td></td>
<td>0.230</td>
<td>0.200</td>
<td>1.1500</td>
<td>0.050</td>
<td>0.031</td>
<td>1.6129</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td>1.783</td>
<td>1.522</td>
<td>1.711</td>
<td></td>
<td>2.003</td>
<td>1.632</td>
<td>2.273</td>
<td>0.281</td>
<td>0.110</td>
<td>2.5545</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300</td>
<td></td>
<td></td>
<td>5.918</td>
<td>5.289</td>
<td>1.1189</td>
<td></td>
<td>7.250</td>
<td>5.278</td>
<td>1.3736</td>
<td>1.122</td>
<td>0.290</td>
<td>3.8690</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
<td>13.759</td>
<td>12.178</td>
<td>1.1298</td>
<td></td>
<td>15.803</td>
<td>12.107</td>
<td>1.3053</td>
<td>2.438</td>
<td>0.691</td>
<td>3.5210</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td></td>
<td></td>
<td>45.245</td>
<td>39.987</td>
<td>1.1315</td>
<td></td>
<td>53.984</td>
<td>40.138</td>
<td>1.3450</td>
<td>8.624</td>
<td>2.043</td>
<td>4.2212</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td>105.251</td>
<td>93.168</td>
<td>1.1297</td>
<td></td>
<td>126.222</td>
<td>97.631</td>
<td>1.2928</td>
<td>18.927</td>
<td>4.347</td>
<td>4.3540</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0.5</td>
<td></td>
<td>0.211</td>
<td>0.211</td>
<td>1.0000</td>
<td></td>
<td>0.211</td>
<td>0.201</td>
<td>1.0498</td>
<td>0.040</td>
<td>0.020</td>
<td>2.0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td>1.692</td>
<td>1.623</td>
<td>1.0425</td>
<td></td>
<td>1.703</td>
<td>1.632</td>
<td>1.0436</td>
<td>0.270</td>
<td>0.110</td>
<td>2.4545</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300</td>
<td></td>
<td></td>
<td>6.621</td>
<td>5.327</td>
<td>1.2429</td>
<td></td>
<td>5.866</td>
<td>5.297</td>
<td>1.1078</td>
<td>0.831</td>
<td>0.361</td>
<td>2.3019</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
<td>12.748</td>
<td>12.057</td>
<td>1.0673</td>
<td></td>
<td>13.561</td>
<td>12.428</td>
<td>1.0912</td>
<td>1.973</td>
<td>0.651</td>
<td>2.3184</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td></td>
<td></td>
<td>42.144</td>
<td>39.497</td>
<td>1.0670</td>
<td></td>
<td>44.805</td>
<td>40.619</td>
<td>1.1031</td>
<td>6.269</td>
<td>2.393</td>
<td>2.6197</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>700</td>
<td></td>
<td></td>
<td>66.075</td>
<td>62.309</td>
<td>1.0604</td>
<td></td>
<td>70.261</td>
<td>64.403</td>
<td>1.0910</td>
<td>9.884</td>
<td>3.595</td>
<td>2.7494</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td>97.775</td>
<td>92.162</td>
<td>1.0609</td>
<td></td>
<td>105.47</td>
<td>95.307</td>
<td>1.1067</td>
<td>14.071</td>
<td>4.867</td>
<td>2.8911</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>2.5</td>
<td></td>
<td>0.291</td>
<td>0.291</td>
<td>1.1594</td>
<td></td>
<td>0.200</td>
<td>0.200</td>
<td>1.0000</td>
<td>0.030</td>
<td>0.030</td>
<td>1.0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td>1.662</td>
<td>1.592</td>
<td>1.0440</td>
<td></td>
<td>1.642</td>
<td>1.592</td>
<td>1.0314</td>
<td>0.110</td>
<td>0.100</td>
<td>1.1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300</td>
<td></td>
<td></td>
<td>5.358</td>
<td>5.258</td>
<td>1.0190</td>
<td></td>
<td>5.358</td>
<td>5.037</td>
<td>1.0637</td>
<td>0.401</td>
<td>0.230</td>
<td>1.7435</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
<td>12.228</td>
<td>11.938</td>
<td>1.0243</td>
<td></td>
<td>12.588</td>
<td>12.067</td>
<td>1.0432</td>
<td>0.781</td>
<td>0.481</td>
<td>1.6337</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td></td>
<td></td>
<td>23.824</td>
<td>23.203</td>
<td>1.0268</td>
<td></td>
<td>24.235</td>
<td>23.444</td>
<td>1.0337</td>
<td>1.502</td>
<td>1.012</td>
<td>1.4842</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td></td>
<td></td>
<td>40.338</td>
<td>39.177</td>
<td>1.0296</td>
<td></td>
<td>41.293</td>
<td>40.088</td>
<td>1.0302</td>
<td>2.754</td>
<td>1.652</td>
<td>1.6671</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>700</td>
<td></td>
<td></td>
<td>63.542</td>
<td>61.719</td>
<td>1.0295</td>
<td></td>
<td>65.224</td>
<td>62.921</td>
<td>1.0366</td>
<td>4.506</td>
<td>2.414</td>
<td>1.8666</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td>94.096</td>
<td>91.211</td>
<td>1.0316</td>
<td></td>
<td>96.779</td>
<td>95.649</td>
<td>1.0118</td>
<td>6.560</td>
<td>3.465</td>
<td>1.8932</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 6: The running time experimental results, comparison between the generic and the dedicated implementations of the new-improved Dynamic Programming method, bounded-#(min+ε) approach.

**Experimental conditions:**
- Algorithm: Dynamic Programming, bounded-#(min+ε) approach
- Distance: Sum of the Euclidean Distance, measured to the line support, computed using the incremental technique
- Input curve: file “lkl1.dat”

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>ε</th>
<th>Running time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Generic</td>
</tr>
<tr>
<td>50</td>
<td>200</td>
<td>12.085813</td>
<td>0.120</td>
</tr>
<tr>
<td>300</td>
<td>29.587154</td>
<td>0.381</td>
<td>0.250</td>
</tr>
<tr>
<td>400</td>
<td>50.601252</td>
<td>0.701</td>
<td>0.491</td>
</tr>
<tr>
<td>500</td>
<td>84.128794</td>
<td>1.152</td>
<td>0.902</td>
</tr>
<tr>
<td>600</td>
<td>110.867001</td>
<td>1.732</td>
<td>1.201</td>
</tr>
<tr>
<td>700</td>
<td>140.939479</td>
<td>2.344</td>
<td>1.723</td>
</tr>
<tr>
<td>800</td>
<td>166.246569</td>
<td>2.995</td>
<td>2.163</td>
</tr>
<tr>
<td>75</td>
<td>200</td>
<td>5.599638</td>
<td>0.140</td>
</tr>
<tr>
<td>300</td>
<td>20.198762</td>
<td>0.501</td>
<td>0.380</td>
</tr>
<tr>
<td>400</td>
<td>33.451724</td>
<td>0.961</td>
<td>0.681</td>
</tr>
<tr>
<td>500</td>
<td>57.598198</td>
<td>1.422</td>
<td>1.081</td>
</tr>
<tr>
<td>600</td>
<td>78.229302</td>
<td>1.993</td>
<td>1.552</td>
</tr>
<tr>
<td>700</td>
<td>101.005928</td>
<td>2.754</td>
<td>2.003</td>
</tr>
<tr>
<td>800</td>
<td>118.372874</td>
<td>3.716</td>
<td>2.704</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>0.769231</td>
<td>0.150</td>
</tr>
<tr>
<td>300</td>
<td>12.584223</td>
<td>0.541</td>
<td>0.401</td>
</tr>
<tr>
<td>400</td>
<td>23.468487</td>
<td>0.951</td>
<td>0.791</td>
</tr>
<tr>
<td>500</td>
<td>41.522231</td>
<td>1.582</td>
<td>1.182</td>
</tr>
<tr>
<td>600</td>
<td>58.055911</td>
<td>2.294</td>
<td>1.752</td>
</tr>
<tr>
<td>700</td>
<td>77.919991</td>
<td>3.195</td>
<td>2.463</td>
</tr>
<tr>
<td>800</td>
<td>91.845391</td>
<td>4.266</td>
<td>3.204</td>
</tr>
</tbody>
</table>

Table 7: The running time experimental results, comparison between the exact computation version and the trigonometric based implementations of the Euclidean Distances assessment algorithm.

**Experimental conditions:**
- Algorithm: Dynamic Programming, bounded-#(min+ε) approach
- Input curve: file “lkl1.dat”
- Distance: Maximum Euclidean Distance, measured to the line support (ls) and measured to the segment (seg)
- Input number of points N=560

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>M</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact Computation</td>
<td>MaxEuclid_LS</td>
<td>36.407</td>
<td>36.814</td>
<td>37.262</td>
<td>37.931</td>
<td>39.034</td>
<td>40.529</td>
<td>41.832</td>
</tr>
<tr>
<td>MaxEuclid_SEG</td>
<td>55.975</td>
<td>57.921</td>
<td>59.532</td>
<td>62.062</td>
<td>63.883</td>
<td>64.623</td>
<td>65.981</td>
<td></td>
</tr>
</tbody>
</table>
Table 8 The running time experimental results measured for implementations of the Dynamic Programming method using different kind of kernels and base data types

**Experimental conditions:**
- Algorithm: Dynamic Programming, bounded-\# (min-c) approach
- Input curve: file "lk1.dat"
- Distances: Sum of the Euclidean Distances
- Output number of points M=64

<table>
<thead>
<tr>
<th>N</th>
<th>M</th>
<th>E</th>
<th>Homogeneous</th>
<th>Cartesian</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>leda_integer</td>
<td>GrpZ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>normal</td>
<td>rescale</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>0.000000</td>
<td>2.964</td>
<td>4.998</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>1.463415</td>
<td>27.039</td>
<td>43.302</td>
</tr>
<tr>
<td>256</td>
<td>64</td>
<td>16.365380</td>
<td>313.996</td>
<td>355.251</td>
</tr>
<tr>
<td>350</td>
<td>64</td>
<td>30.748217</td>
<td>922.290</td>
<td>949.887</td>
</tr>
<tr>
<td>450</td>
<td>64</td>
<td>52.350310</td>
<td>2285.697</td>
<td>2204.664</td>
</tr>
<tr>
<td>560</td>
<td>64</td>
<td>80.657475</td>
<td>5995.215</td>
<td>3864.191</td>
</tr>
</tbody>
</table>

Table 9 The running time experimental results comparison between different kind of distances measured for 2-dimensional and 3-dimensional implementations

**Experimental conditions:**
- Algorithm: Dynamic Programming, bounded-\# (min-c) approach
- Input curve: file "lk1.dat"
- Input number of points N= 640

<table>
<thead>
<tr>
<th>Distance</th>
<th>M</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxEuclidean</td>
<td>2D</td>
<td>36.407</td>
<td>36.814</td>
<td>37.262</td>
<td>37.931</td>
<td>39.034</td>
<td>40.529</td>
<td>41.832</td>
</tr>
<tr>
<td>MaxInfinity</td>
<td>2D</td>
<td>38.709</td>
<td>40.102</td>
<td>40.829</td>
<td>41.737</td>
<td>42.821</td>
<td>44.329</td>
<td>46.257</td>
</tr>
<tr>
<td>MaxManhattan</td>
<td>2D</td>
<td>37.429</td>
<td>38.686</td>
<td>39.524</td>
<td>40.641</td>
<td>42.071</td>
<td>43.824</td>
<td>45.663</td>
</tr>
<tr>
<td>MaxEuclidean</td>
<td>3D</td>
<td>105.6185</td>
<td>106.792</td>
<td>108.0989</td>
<td>110.0997</td>
<td>113.2395</td>
<td>117.5766</td>
<td>121.3966</td>
</tr>
<tr>
<td>MaxInfinity</td>
<td>3D</td>
<td>108.6593</td>
<td>112.5995</td>
<td>114.6103</td>
<td>117.1591</td>
<td>120.202</td>
<td>124.4351</td>
<td>129.8471</td>
</tr>
<tr>
<td>MaxManhattan</td>
<td>3D</td>
<td>108.7654</td>
<td>112.4181</td>
<td>114.8533</td>
<td>118.0992</td>
<td>122.2546</td>
<td>127.3487</td>
<td>132.7799</td>
</tr>
</tbody>
</table>

60
Table 10: The running time experimental results comparison between different kind of polygonal approximation algorithms for the bounded-ε approach, using the sum of the Euclidean distances as error assessment.

**Experimental conditions:**
- Algorithms: bounded-ε approach
- Distance: Sum of the Euclidean Distances
- Input curves: file “kkl.dat”

<table>
<thead>
<tr>
<th>Err N</th>
<th>Mopt</th>
<th>DynProg</th>
<th>DynProg / incr</th>
<th>Graph Search</th>
<th>RecSplit</th>
<th>Mrs-Mopt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>NEW</td>
<td>NEW_IMP</td>
<td>NEW</td>
<td>NEW_IMP</td>
<td>time</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>63</td>
<td>0.230</td>
<td>0.200</td>
<td>0.050</td>
<td>0.031</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>100</td>
<td>2.003</td>
<td>1.632</td>
<td>0.281</td>
<td>0.110</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>150</td>
<td>7.250</td>
<td>5.278</td>
<td>1.122</td>
<td>0.290</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>195</td>
<td>15.803</td>
<td>12.107</td>
<td>2.433</td>
<td>0.691</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>243</td>
<td>31.145</td>
<td>23.454</td>
<td>5.057</td>
<td>1.231</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>290</td>
<td>53.984</td>
<td>40.138</td>
<td>8.624</td>
<td>2.043</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>345</td>
<td>87.876</td>
<td>62.991</td>
<td>13.299</td>
<td>3.065</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>367</td>
<td>126.222</td>
<td>97.631</td>
<td>18.927</td>
<td>4.347</td>
</tr>
</tbody>
</table>

| 10    | 100  | 25  | 0.211 | 0.201 | 0.040 | 0.020 | 0.010 | 0.010 | 38 | 0.000 | 53 | 28 |
|       | 200  | 57  | 1.703 | 1.632 | 0.270 | 0.110 | 0.070 | 0.060 | 75 | 0.000 | 108 | 51 |
|       | 300  | 111 | 5.858 | 5.297 | 0.831 | 0.361 | 0.281 | 0.180 | 134 | 0.000 | 182 | 71 |
|       | 400  | 147 | 13.561 | 12.428 | 1.973 | 0.851 | 0.621 | 0.250 | 170 | 0.010 | 244 | 97 |
|       | 500  | 195 | 26.258 | 23.974 | 3.706 | 1.502 | 1.292 | 0.490 | 218 | 0.010 | 307 | 112 |
|       | 600  | 242 | 44.805 | 40.619 | 6.269 | 2.393 | 1.932 | 0.661 | 265 | 0.010 | 382 | 140 |
|       | 700  | 295 | 70.261 | 64.403 | 9.884 | 3.595 | 3.054 | 0.881 | 320 | 0.020 | 449 | 154 |
|       | 800  | 317 | 105.47 | 95.307 | 14.071 | 4.867 | 4.216 | 1.081 | 342 | 0.020 | 475 | 158 |

| 100   | 100  | 4   | 0.200 | 0.200 | 0.030 | 0.030 | 0.010 | 0.010 | 4 | 0.000 | 5 | 1 |
|       | 200  | 9   | 1.642 | 1.592 | 0.110 | 0.100 | 0.060 | 0.060 | 9 | 0.000 | 14 | 5 |
|       | 300  | 12  | 5.358 | 5.037 | 0.401 | 0.230 | 0.160 | 0.140 | 24 | 0.000 | 27 | 15 |
|       | 400  | 21  | 12.588 | 12.067 | 0.781 | 0.481 | 0.430 | 0.250 | 60 | 0.000 | 52 | 31 |
|       | 500  | 41  | 24.235 | 23.444 | 1.502 | 1.012 | 0.681 | 0.481 | 108 | 0.010 | 100 | 59 |
|       | 600  | 88  | 41.299 | 40.088 | 2.754 | 1.652 | 1.161 | 0.681 | 155 | 0.010 | 163 | 110 |
|       | 700  | 76  | 65.224 | 62.921 | 4.506 | 2.414 | 1.923 | 0.851 | 210 | 0.010 | 233 | 157 |
|       | 800  | 91  | 96.779 | 95.649 | 6.560 | 3.465 | 2.704 | 1.051 | 232 | 0.010 | 259 | 168 |
**Table 11** The running time experimental results: comparison between different kind of polygonal approximation algorithms for the bounded-ε approach, using the maximum of the Euclidean distances as error assessment

**Experimental conditions:**
- Algorithm: bounded-ε approach
- Distance: Maximum of the Euclidean Distances
- Input curve: file "cik1.dat"

<table>
<thead>
<tr>
<th>Err</th>
<th>N</th>
<th>Mopt</th>
<th>DynProg</th>
<th>Graph Search</th>
<th>RecSplit</th>
<th>Mrs-Mopt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>NEW</td>
<td>NEW-JMP</td>
<td>time [s]</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>100</td>
<td>67</td>
<td>0.211</td>
<td>0.211</td>
<td>0.140</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>104</td>
<td>1.783</td>
<td>1.522</td>
<td>0.701</td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>163</td>
<td>5.918</td>
<td>5.289</td>
<td>1.492</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>199</td>
<td>13.759</td>
<td>12.178</td>
<td>2.865</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>247</td>
<td>26.489</td>
<td>23.464</td>
<td>4.577</td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>294</td>
<td>45.245</td>
<td>39.987</td>
<td>6.399</td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>349</td>
<td>71.137</td>
<td>62.951</td>
<td>8.762</td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>371</td>
<td>105.251</td>
<td>93.168</td>
<td>11.487</td>
<td>0.010</td>
</tr>
<tr>
<td>0.5</td>
<td>100</td>
<td>21</td>
<td>0.211</td>
<td>0.211</td>
<td>0.140</td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>31</td>
<td>1.692</td>
<td>1.623</td>
<td>0.721</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>39</td>
<td>6.621</td>
<td>5.327</td>
<td>1.622</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>56</td>
<td>12.748</td>
<td>12.057</td>
<td>2.844</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>76</td>
<td>24.669</td>
<td>23.284</td>
<td>4.596</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>89</td>
<td>42.144</td>
<td>39.497</td>
<td>6.541</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>100</td>
<td>66.075</td>
<td>62.309</td>
<td>8.693</td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>110</td>
<td>97.775</td>
<td>92.162</td>
<td>11.196</td>
<td>0.010</td>
</tr>
<tr>
<td>2.5</td>
<td>100</td>
<td>5</td>
<td>0.291</td>
<td>0.251</td>
<td>0.161</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>10</td>
<td>1.662</td>
<td>1.592</td>
<td>0.701</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>12</td>
<td>5.358</td>
<td>5.258</td>
<td>1.562</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>18</td>
<td>12.228</td>
<td>11.938</td>
<td>2.864</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>24</td>
<td>23.824</td>
<td>23.203</td>
<td>4.597</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>29</td>
<td>40.398</td>
<td>39.177</td>
<td>6.681</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>32</td>
<td>63.542</td>
<td>61.719</td>
<td>8.963</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>34</td>
<td>94.096</td>
<td>91.211</td>
<td>11.597</td>
<td>0.000</td>
</tr>
</tbody>
</table>

62
Table 12: The running time experimental results: comparison between different kinds of polygonal approximation algorithms for the bounded approach, using the sum of the Euclidean distances as error assessment.

**Experimental conditions:**
- Algorithm: bounded approach
- Distance: Sum of the Euclidean Distances
- Input curve file: "kk1.dat"

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>NEW</td>
<td>NEW_JUMP</td>
<td></td>
<td>time [s]</td>
<td>Enr</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>---------</td>
<td>-----------------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>50</td>
<td>200</td>
<td>12.085813</td>
<td>1.722</td>
<td>1.753</td>
<td>0.181</td>
<td>0.120</td>
<td>0.080</td>
</tr>
<tr>
<td>300</td>
<td>29.987154</td>
<td>5.648</td>
<td>5.048</td>
<td>0.571</td>
<td>0.381</td>
<td>0.250</td>
<td>0.010</td>
</tr>
<tr>
<td>400</td>
<td>50.601252</td>
<td>12.668</td>
<td>12.448</td>
<td>1.062</td>
<td>0.701</td>
<td>0.491</td>
<td>0.000</td>
</tr>
<tr>
<td>500</td>
<td>84.126794</td>
<td>24.245</td>
<td>23.424</td>
<td>1.723</td>
<td>1.152</td>
<td>0.902</td>
<td>0.010</td>
</tr>
<tr>
<td>600</td>
<td>110.867001</td>
<td>41.249</td>
<td>39.857</td>
<td>2.594</td>
<td>1.732</td>
<td>1.201</td>
<td>0.010</td>
</tr>
<tr>
<td>700</td>
<td>140.939479</td>
<td>64.593</td>
<td>62.411</td>
<td>3.906</td>
<td>2.344</td>
<td>1.723</td>
<td>0.000</td>
</tr>
<tr>
<td>800</td>
<td>166.246569</td>
<td>95.878</td>
<td>92.222</td>
<td>5.648</td>
<td>2.995</td>
<td>2.163</td>
<td>0.010</td>
</tr>
<tr>
<td>75</td>
<td>200</td>
<td>5.599638</td>
<td>1.723</td>
<td>1.523</td>
<td>0.201</td>
<td>0.140</td>
<td>0.090</td>
</tr>
<tr>
<td>300</td>
<td>20.197672</td>
<td>5.718</td>
<td>5.468</td>
<td>0.731</td>
<td>0.501</td>
<td>0.380</td>
<td>0.000</td>
</tr>
<tr>
<td>400</td>
<td>33.451724</td>
<td>12.918</td>
<td>12.388</td>
<td>1.461</td>
<td>0.961</td>
<td>0.681</td>
<td>0.010</td>
</tr>
<tr>
<td>500</td>
<td>57.598198</td>
<td>24.936</td>
<td>23.754</td>
<td>2.504</td>
<td>1.422</td>
<td>1.081</td>
<td>0.000</td>
</tr>
<tr>
<td>600</td>
<td>78.229302</td>
<td>42.589</td>
<td>40.338</td>
<td>4.136</td>
<td>1.993</td>
<td>1.552</td>
<td>0.010</td>
</tr>
<tr>
<td>700</td>
<td>101.059286</td>
<td>66.766</td>
<td>62.851</td>
<td>6.109</td>
<td>2.754</td>
<td>2.003</td>
<td>0.010</td>
</tr>
<tr>
<td>800</td>
<td>118.372874</td>
<td>99.192</td>
<td>92.864</td>
<td>8.232</td>
<td>3.716</td>
<td>2.704</td>
<td>0.010</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>0.769231</td>
<td>1.773</td>
<td>1.542</td>
<td>0.231</td>
<td>0.150</td>
<td>0.090</td>
</tr>
<tr>
<td>300</td>
<td>12.084223</td>
<td>5.678</td>
<td>5.318</td>
<td>0.731</td>
<td>0.541</td>
<td>0.401</td>
<td>0.000</td>
</tr>
<tr>
<td>400</td>
<td>23.468487</td>
<td>13.711</td>
<td>12.668</td>
<td>2.013</td>
<td>0.951</td>
<td>0.791</td>
<td>0.010</td>
</tr>
<tr>
<td>500</td>
<td>41.522231</td>
<td>26.067</td>
<td>23.674</td>
<td>3.575</td>
<td>1.582</td>
<td>1.182</td>
<td>0.010</td>
</tr>
<tr>
<td>600</td>
<td>58.055911</td>
<td>44.013</td>
<td>40.368</td>
<td>5.819</td>
<td>2.294</td>
<td>1.752</td>
<td>0.000</td>
</tr>
<tr>
<td>700</td>
<td>77.919991</td>
<td>68.759</td>
<td>63.351</td>
<td>8.091</td>
<td>3.195</td>
<td>2.463</td>
<td>0.010</td>
</tr>
<tr>
<td>800</td>
<td>91.845391</td>
<td>100.985</td>
<td>93.385</td>
<td>10.785</td>
<td>4.266</td>
<td>3.204</td>
<td>0.010</td>
</tr>
</tbody>
</table>
Table 23 The running time experimental results comparison between different kind of polygonal approximation algorithms for the bounded-\# approach, using the maximum of the Euclidean distances as error assessment.

**Experimental conditions:**
- Algorithms: bounded-\# approach
- Distance: Maximum of the Euclidean Distances
- Input curve file "lkl.dat"

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Exp</th>
<th>DynProg</th>
<th>Graph Search</th>
<th>RecSplit</th>
<th>Ens-Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>NEW</td>
<td>NEW_IMP</td>
<td>time [s]</td>
<td>Mos [s]</td>
</tr>
<tr>
<td>50</td>
<td>200</td>
<td>0.248731</td>
<td>1.743</td>
<td>1.583</td>
<td>0.681</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>0.401639</td>
<td>5.555</td>
<td>5.387</td>
<td>1.742</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>0.675676</td>
<td>12.718</td>
<td>12.388</td>
<td>3.144</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.961538</td>
<td>24.345</td>
<td>23.354</td>
<td>5.007</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>1.000000</td>
<td>41.039</td>
<td>39.647</td>
<td>7.101</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>1.123596</td>
<td>64.803</td>
<td>62.231</td>
<td>9.724</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>1.212871</td>
<td>96.001</td>
<td>91.938</td>
<td>12.559</td>
<td>50</td>
</tr>
<tr>
<td>75</td>
<td>200</td>
<td>0.200000</td>
<td>1.883</td>
<td>1.682</td>
<td>0.771</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>0.247525</td>
<td>6.159</td>
<td>5.328</td>
<td>1.713</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>0.400000</td>
<td>13.961</td>
<td>12.368</td>
<td>3.164</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.527835</td>
<td>26.388</td>
<td>23.684</td>
<td>5.218</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>0.752941</td>
<td>45.676</td>
<td>39.927</td>
<td>7.561</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>0.844828</td>
<td>70.741</td>
<td>62.501</td>
<td>10.014</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>0.900000</td>
<td>105.261</td>
<td>92.493</td>
<td>13.139</td>
<td>73</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>0.200000</td>
<td>1.832</td>
<td>1.672</td>
<td>0.661</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>0.248276</td>
<td>5.666</td>
<td>5.448</td>
<td>1.742</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>0.400000</td>
<td>13.761</td>
<td>12.578</td>
<td>3.185</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.500000</td>
<td>26.187</td>
<td>23.754</td>
<td>5.038</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>0.500000</td>
<td>44.183</td>
<td>40.288</td>
<td>7.201</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>0.500000</td>
<td>72.998</td>
<td>62.881</td>
<td>10.294</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>0.634518</td>
<td>106.914</td>
<td>92.994</td>
<td>13.119</td>
<td>100</td>
</tr>
</tbody>
</table>
References


